

KUMA

**BC
BASIC**

for the

Commodore 64



CONTENTS

Introduction 2

1. Basic graphics 3

2. User-defined graphics 8

3. Basic sprites16

4. Basic Input/Output24

5. A first look at structured programming26

6. High-resolution graphics29

7. Basic sound40

8. Advanced sound47

9. Advanced sprites55

10. More structured programming62

11. More I/O64

12. Advanced character graphics70

13. Machine-code programming aids73

14. Miscellaneous useful facilities77

Index and quick-reference table79

BC BASIC was developed by BC COMPUTERS, 31A Grosvenor Avenue, Long Eaton, Nottingham NG10 3FQ. This product is COPYRIGHT (C) Brian Candler 1983. Any attempt to copy the cartridge or manual is ILLEGAL and may result in prosecution.

This software is supplied in the belief that it operates as specified but neither the author nor the supplier will be held responsible for any problems arising in or from its use.

Published by:-
Kuma Computers Ltd.,
12 Horseshoe Park,
Pangbourne,
Berks RG8 7JW
07357-4335

INTRODUCTION

Welcome to the world of BC BASIC programming. This package will enable you to get the most from your Commodore 64, whether you are a computing expert or just a beginner. BC BASIC makes the 64's advanced facilities, such as hi-res graphics, sprites and sound, easy and simple to use by means of many fast and powerful commands and functions added to the BASIC language. Whatever you use your 64 for, from writing games to serious applications, BC BASIC will help you do it much more easily and quickly.

This manual is arranged so that it begins with the simpler features of BC BASIC, and then progresses to more complex ones. To get the most from BC BASIC, work through the manual, trying out the examples and programs of your own so that you can see for yourself how each command or function works. At the end of each chapter, you may like to try to write some programs using what you have learnt so far. You may be quite surprised by the standard of programs you will be writing after a short while! Even advanced programmers are advised to read through the chapters in sequence, as later chapters will refer to commands and functions explained earlier on.

Now it just remains for me to wish you every success with your BC BASIC programming.

Brian Candler,
author of BC BASIC,

INSERTING THE CARTRIDGE

1. TURN THE MACHINE OFF.
2. Insert the cartridge into the cartridge port - this is the large rectangular hole on the back of the 64. Press it home firmly.
3. Now turn the 64 on. After a couple of seconds, the BC BASIC startup message will appear. BC BASIC is now ready for use.

REMEMBER -NEVER INSERT OR REMOVE THE CARTRIDGE WITH THE MACHINE SWITCHED ON, OR YOU WILL RISK DAMAGING THE CARTRIDGE, YOUR COMMODORE 64, OR BOTH.

Chapter 1

BASIC GRAPHICS

Now you have got BC BASIC up and running, here are a few simple but useful commands to get you started. Try them out at the keyboard.

BORDER col

This changes the border colour of the screen to 'col', where col is in the range 0-15 and represents one of the following colours:

0	Black	8	Orange
1	White	9	Brown
2	Red	10	Light red
3	Cyan	11	Dark grey
4	Purple	12	Medium grey
5	Green	13	Light green
6	Blue	14	Light blue
7	Yellow	15	Light grey

'col' does not have to be a number - it may be a variable or expression, like B or C*2+3. However, if it is outside the range 0-15 an ?ILLEGAL QUANTITY error will result.

Whenever you see 'col' again in a BC BASIC command definition, you know it will mean a number or expression in the range 0-15.

Try typing BORDER 13 - this will change the border of the screen from its normal medium-grey colour to light green. Pressing RUN/STOP and RESTORE together will reset the screen colours to a dark grey (11) background, with a medium grey (12) border and cyan (3) letters.

PAPER col

This command, similar to BORDER above, changes the screen background colour (the 'paper' on which the computer writes) to col.

BORDER and PAPER are the same as POKE 53280, col and POKE 53281,col respectively, but much easier to remember!

INK col

This command sets what colour is to be printed to the screen. For example, typing INK 14 makes "READY." and all further characters appear as light blue, rather than cyan.

Try the following program, which displays all of the colours available on the 64:

```
10 FOR C=0 TO 15
20 INK C
30 PRINT "COMMODORE 64!"
40 NEXT C
```

AT y,x

This command instantly moves the cursor to the xth position along line y. It makes moving characters and positioning titles etc. easy. y must be in the range 0 to 24, and x must be in the range 0 to 39. 0,0 is the top left corner of the screen. y comes before x for two reasons:

- a). For convenience (in being able to specify the line before the column)
- b). For compatibility with BASICs on other machines.

AT may be used on its own (ready for PRINTing) or actually inside a print statement:

e.g. PRINT AT 3,0;"HI";AT 5,4;"THERE!"

THE SAME APPLIES TO BORDER, PAPER AND INK. INK is quite commonly used inside a PRINT statement.

Short forms for keywords

You may have met abbreviations for commands and functions in standard Commodore BASIC - e.g. ? for PRINT and G shift-O for GOTO. Many BC BASIC commands can be abbreviated in a shortened form. Of those you have met so far:

BORDER may be abbreviated to B shift-O

PAPER may be abbreviated to P shift-A

From now on, if a command or function has a short form, it will be printed below the definition in brackets, like this:

BORDER col
(bO)

where a capital O indicates that it must be shifted.

Animation using PRINT AT

This is far easier than anything you may have tried using POKE. Just follow these steps:

1. Put a character on the screen at a certain position (using AT)
2. Wait for a moment.
3. Print a space at the same place as the character (this wipes it off)
4. Change its position and go back to step 1.

For example, to move a star across the screen:

```
10 Y=5:X=0:REM FIRST CHARACTER ON LINE 5
20 PRINT AT Y,X;"*"
30 FOR K=1 TO 50:NEXT K:REM A SHORT DELAY
40 PRINT AT Y,X;" "
50 X=X+1:REM MOVE IT ACROSS
60 IF X<40 THEN GOTO 20
```

This program prints a star at 5,0, waits a moment, wipes it off, and then adds 1 to X (so it will next be displayed at 5,1 instead of 5,0) and carries on until it reaches the right-hand side of the screen, when it stops.

Some improvements

5 PRINT "[cls]" will clear the screen at the beginning of the program. [cls] appears as a reverse-heart symbol, and is obtained by pressing SHIFT and CLR/HOME together.

Changing the number '50' in line 30 will alter the speed of movement - the smaller the number, the less the delay, and so the faster the star will move. If you make the delay too small, however, the character will be wiped off the screen almost as soon as it was put there, and so the star will flicker appreciably.

Try changing the character printed in line 20. You can print something longer than 1 character (as long as there are the same number of spaces in line 40) so you could even have your name moving across the screen!

Of course, as well as being able to put characters on the screen, it would be nice to be able to determine what character is at a certain position on the screen. The following definitions are for functions, as opposed to commands - they may only be used within expressions (e.g. LET A=GCOL(L,C))

SCR(y,x) This function returns the 'POKE code' of the character at a certain screen location y,x: this is the code you would POKE to screen memory to make that character appear. There is a table of POKE codes (or 'screen display codes') in the Commodore 64 User Manual - however, the function CODE, described below, will find them for you. Like AT, y and x are numeric expressions where y is in the range 0-24 and x in the range 0-39. An example using SCR will follow shortly.

GCOL(y,x) This function, which has the same format as SCR, (gC) returns the colour of the character at y,x, as a number 0-15. GCOL stands for Graphics COLOUR.

e.g. if PRINT GCOL (3,5) prints 14, then the character on line 3, column 5 is light-blue.

CODE(n) This function is used to convert characters to
or CODE(n\$) their POKE codes. If there is a number or numeric
(coD) expression in the brackets, this is taken as an ASCII code and converted to its associated POKE code. If n is not a printable ASCII character, i.e. if it is out of the range 0-255 or is a control character, then an ?ILLEGAL QUANTITY error is printed.

The second format shown, with a string inside the brackets, converts the first character of the string to its POKE code, unless the first character is a [rvs] or [rvs-off] character, in which case the second character is converted, and 128 added to the code if the first character was [rvs]. Again, if the character does not have a POKE code, an ?ILLEGAL QUANTITY error will result.

A few examples should clear things up:

```
PRINT CODE(65) prints 1, since 65 is the ASCII
                code for an 'A', and 1 is the
                POKE code for this character.
A=CODE("Z")    Sets A to 26
A=CODE("[rvs]Z") Sets A to 154 (=26+128)
PRINT CODE(17) Gives an error, since 17 is a
                control code, not a printable
                character.
```

An example of the use of SCR - collision detection

If you use SCR just before PRINT AT, you can find out what character you are printing on top of. We can now modify the 'moving star' program so that it crashes against a mountain.

```

10 PRINT "[cls]"
20 PRINT AT 7,0;"          XXX"
30 PRINT AT 8,0;"    XXX          XXX"
40 PRINT AT 9,0;"XXXXXXXXXXXXXXXXXXXXXXXXXXXX"
50 P=0:REM P IS POSITION ALONG LINE
60 IF SCR(7,P)<>CODE(" ") THEN GOTO 120:REM CRASH?
70 PRINT AT 7,P;"*"
80 FOR K=1 TO 50:NEXT K
90 PRINT AT 7,P;" "
100 P=P+1:REM MOVE IT ALONG
110 GOTO 60
120 PRINT AT 7,P-2;"CRASH!"
130 AT 20,0:END

```

Lines 10-40 of this program clear the screen and display a terrain. Replace the Xs with graphic symbols - chequered flags perhaps (press the Commodore and '+' keys together).

Line 50 sets the initial position along the line.

Line 60 is the important one. It says "if the character where I'm just about to put a star is not a space, then goto line 120" - in other words, when the star reaches the terrain, it stops moving and the program continues on line 120.

Lines 70-110 are the same as the old moving star program, except without the test to see if we are at the edge of the screen (it is unnecessary, since we are going to crash first).

Line 120 prints CRASH! where the star was, and line 130 moves the cursor to the bottom of the screen before stopping.

In the next chapter, we shall find out how to overcome the limitations of only having the graphics available on the keyboard, by being able to define your own shapes.

Exercise - try bouncing a character backwards and forwards (hint - split the program into two parts, one to move left to right and the other to move back again).

Chapter 2

USER-DEFINED GRAPHICS

These are a powerful feature of the 64 which allow you to create your own characters and symbols. Firstly, however, we must digress a little.

Binary and hexadecimal notation

BINARY, or Base 2, is the system of numbering which the computer uses. It is ideal for this purpose, since it only consists of 0s and 1s, and can be stored in the computer as 'Off' or 'On'.

1 binary digit = 1 'bit'

1 byte = 8 bits. Each memory location in the computer is 1 byte wide. 1K, or kilobyte, equals 1024 bytes.

BC BASIC uses the % function to convert binary to decimal. So:

```
PRINT %1101
```

prints 13 (13=8+4+1).

HEXADECIMAL, or just HEX for short, is a convenient way of representing computer numbers without strings of 0s and 1s. It is actually base 16 - each digit may be 0-9 or A-F. Hex numbers are preceded by the '\$' sign.

e.g.

```
PRINT $4D2
```

prints 1234 (1234=4*256+13*16+2)

Don't worry if this all seems confusing - it is not vital to understand. However, you should know that memory addresses range from \$0000 to \$FFFF - a total of 65536 bytes.

Here is a 'memory map' - a chart of how the memory is arranged - on a Commodore 64 under BC BASIC, as the 6510 processor chip sees it. Again, do not worry if you do not understand it all - however, note the position of USER RAM as this is important.

: \$E000-\$FFFF	: 8K KERNAL ROM	:
: \$D000-\$DFFF	: 4K INPUT/OUTPUT (optionally 4K CHARACTER GENERATOR ROM)	:
: \$C000-\$CFFF	: Part of BC BASIC in RAM	:
: \$A000-\$BFFF	: 8K BASIC ROM	:
: \$8000-\$9FFF	: 8K BC BASIC ROM	:
: \$0800-\$7FFF	: 30K USER RAM for programs and data	:
: \$0400-\$07FF	: Normal position of SCREEN RAM and Sprite pointers	:
: \$0000-\$03FF	: BASIC reserved RAM.	:

This, on the other hand, is how the VIC chip sees the 64's memory:

: \$C000-\$FFFF	: BLOCK 3	: RAM.	:
: \$8000-\$BFFF	: BLOCK 2	: \$A000-\$BFFF RAM	:
:	:	: \$9800-\$9FFF CHARACTER ROM	:
:	:	: (Lower case).	:
:	:	: \$9000-\$97FF CHARACTER ROM	:
:	:	: (Upper case).	:
:	:	: \$8000-\$8FFF RAM.	:
: \$4000-\$7FFF	: BLOCK 1	: RAM	:
: \$0000-\$3FFF	: BLOCK 0	: \$2000-\$3FFF RAM.	:
:	:	: \$1800-\$1FFF CHARACTER ROM	:
:	:	: (Lower case)	:
:	:	: \$1000-\$17FF CHARACTER ROM	:
:	:	: (Upper case)	:
:	:	: \$0000-\$0FFF RAM.	:

There are several very important points to notice on this diagram.

Firstly, the VIC chip cannot 'see' the BASIC and KERNAL ROMs - it just sees the RAM which sits behind them. This memory is used by BC BASIC for hi-res information, so as not to waste BASIC memory.

Secondly, notice that the diagram is divided into four 16K BLOCKS. THE VIC CHIP CAN ONLY ACCESS ONE OF THESE BLOCKS AT A

TIME. Therefore, all graphics, screen data and sprites must be in the same 16K block - if you wish to move one of them out into a different block, you must move all the rest at the same time too.

Thirdly, the VIC chip cannot see RAM at \$1000-\$1FFF or \$9000-\$9FFF - it sees the character generator ROM instead. Therefore, you cannot put your user-defined graphics data there (you would be overwriting your BASIC program anyway!) so the user-defined graphics commands will return an error if you try to; you will have to move the character set first. All will be explained shortly.

User-defined graphics - step-by-step

Basically, this is an all-or-nothing process on the Commodore 64. You must tell the VIC chip to get its character data from a different area of RAM, copy the ROM characters there and then change the ones you want.

a). Where do I put the character set?

To begin with, it would obviously be simplest if we were to keep the VIC chip in the same 16K block that it is used to - block 0, which is at \$0000-\$3FFF (see diagram above). A whole character set takes 2K, or \$800, bytes. Therefore, we shall put the character set at \$3800-\$3FFF and tell BASIC not to use any memory above \$37FF. This obviously wastes memory, but it is simplest to begin with, and the 12K which is left is sufficient for programs of a reasonable length. Later in this chapter we shall see how to put the character set right at the top of user RAM.

b). What do I do next?

Firstly, we must tell BASIC that the RAM from \$3800 upwards is reserved, and not to be interfered with. This is done by the following command:

```
HIMEM=$3800
(hI)
```

Please note that this command clears all variables - therefore it should be the first line of a program, or should be typed in immediate mode (directly at the keyboard), when the contents of variables do not matter. Try it now.

Now we must tell VIC where to get its characters from - this is done using the CHARSET command. To find what number should follow CHARSET, take the address of the character set and divide it by 2048 (\$800) - in our case, this is \$3800/\$800, which turns out to be 7 (try it for yourself).

N.B. Your character set MUST start on a 2K boundary - i.e. it must start at \$0000, \$0800, \$1000, \$1800, \$2000 etc.

Now you can type the following command to move the character set:

CHARSET 7
(chA)

Oh dear! The screen seems to have turned into a dreadful mess. Do not worry - we haven't copied the ROM characters into RAM yet, remember?
Typing the following remedies the situation:

COPY 0 TO 255
(coP)

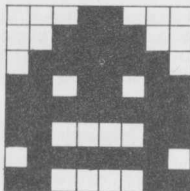
It will now seem that you are in normal mode with only 12K of memory free - but you are now ready for the next stage.

HINT: To copy your characters from the lower case (small letters) character set instead of the upper case one, use instead:

COPY £1,0 TO 255 [COPY £0 may be used instead of
COPY - i.e. COPY £0,0 TO 255 =
COPY 0 TO 255]

c). Defining your own characters

This is the fun part! Firstly, draw out your character as an 8 by 8 square matrix, as follows:



A coloured-in block represents a dot on the screen 'ON' or illuminated in the character colour - a blank square represents a screen dot in the background or PAPER colour. You may find graph-paper useful for this operation. You are recommended to make all vertical lines in your character at least 2 dots wide to help prevent 'chroma noise' (colour distortion) on your TV. This is not necessary if you are using a black-and-white TV.

Now decide which character you are going to substitute the 'Space Invader' or whatever for - let's say the '*'. You must first find the POKE code of the character, using the CODE function, as described in the previous chapter. Typing:

PRINT CODE("*")

will give the required number - 42 in this case. Now you can define your character in rows, using '0' for a space and '1' for a dot on the screen:

```
DEFUSR 42;0;%00011000
DEFUSR 42;1;%00111100
DEFUSR 42;2;%01111110
DEFUSR 42;3;%11011011
DEFUSR 42;4;%11111111
DEFUSR 42;5;%11000011
DEFUSR 42;6;%01111110
DEFUSR 42;7;%11000011
(dEuS)
```

From now on, every time you type a '*', a space-invader, or whatever character you defined, will appear. The number sandwiched between semicolons (and they must be semicolons) is the row of the character being defined. The % indicates binary, and converts the binary digits following it to decimal. Each row is 8 bits, or 1 byte.

The change is actually an illusion - using a '*' in BASIC programs will work as normal, although it will look odd. If you want to copy the normal '*' back from ROM, to erase your character, use the COPY command:

COPY 42

To return to normal ROM characters, type CHARSET 2
To switch to lower-case ROM characters, type CHARSET 3
(this is because 2*\$800=\$1000 and 3*\$800=\$1800, and if you refer back to the 64's memory map as VIC sees it, this is the position of the ROM character set).

After this, to reset the top of memory pointer (so BASIC can use all its RAM again) type:

```
HIMEM=$8000
```

The DEFUSR command is more flexible than shown above - it can take more than one row of data at a time (separate the rows by commas) and, if no row number is specified, 0 is assumed. So:

```
DEFUSR 42,%00011000,%00111100,%01111110,%11011011
DEFUSR 42;4;%11111111,%11000011,%01111110,%11000011
```

has the same effect as the 8-line character definition for the space-invader given previously. Note that you cannot go past row 7 when using DEFUSR.

Some examples:

```
DEFUSR 81,56,120,199,192,199,120,56,0
```

defines code 81 (a shift-Q) as a Commodore logo.

```
10 FOR R=0 TO 7
20 READ A$:A=VAL("%"+A$)
30 DEF USR 131;R;A
40 NEXT R
50 DATA 001111100,01000010,10011001,10100001
60 DATA 10100001,10011001,01000010,00111100
```

This defines code 131 (reverse-C) as a Copyright symbol. Notice in line 20 that you can use % inside VAL, to convert binary strings to decimal (the same applies to \$) and in line 30 that DEFUSR can be split into two words.

Reading the character set

It is useful to be able to read the definitions of characters - this is performed by the following function.

CHARSET(ch,rw)
(chA) This function fetches row 'rw' of character 'ch' in the current character set, and returns it as a value in the range 0-255, where each bit of the number corresponds to a dot in the character. 'ch' must be between 0 and 255, and 'rw' must be between 0 and 7. 'ch' must be the POKE code (obtained by the CODE function) of the required character.

The next function is useful in the context of reading and displaying characters, as it converts decimal to binary.

NOTE: IN BC BASIC command and function definitions, anything in square brackets [] is optional - it may be left out if desired.

BIN\$(n[,d[,x,y]])
(bI) This function returns the number 'n', which may be between -\$FFFFFFF and +\$FFFFFFF, as a binary string. If n is positive, the first character of the string is a space; if n is negative, it is a minus sign (just like STR\$). If d is specified (in the range 0-32) it is the minimum number of digits which will be returned - padded with zeros if necessary. If x and y are specified (each must be between 0 and 255) they are the ASCII codes which will be returned instead of the 0 and 1 characters respectively - i.e. you can substitute characters of your own choice for the 0 and 1 symbols.

```
e.g. PRINT BIN$(5)      prints 101
A$=BIN$(53,8)      sets A$ to " 00110101"
PRINT BIN$(53,12,46,42)
                     prints .....**.*.*
                     (since 46 is the code for '.' and 42 is
                     the code for '*').
```

Here is an example program using CHARSET and BIN\$. It prints the 64's character set continuously as large (8 by 8) characters.

```
10 PRINT"[cls]"
20 FOR C=0 TO 255:REM DISPLAY ALL CHARACTERS
30 AT 0,0:REM MOVE CURSOR TO TOP OF SCREEN
40 FOR R=0 TO 7:REM R IS A ROW COUNTER
50 PRINT BIN$(CHARSET(C,R),8,46,209)
60 NEXT R:REM PRINT NEXT ROW
70 NEXT C:REM NEXT CHARACTER
80 GOTO 20
```

Line 50 is the vital line in this program. It says:

"Print in binary the Rth row of character C in the character set, as an 8-digit binary number, printing character code 46 (a full stop) for a 0 and character code 209 (a shift-Q symbol) for a 1"

Moving VIC into a different 16K block

If you try to use CHARSET followed by a number greater than 7, you will get an ?ILLEGAL QUANTITY error - this is because $8 \times \$800 = \4000 - i.e. you have tried to move the character set into the next 16K block. In fact, this is possible, but only if you move the screen RAM (the area where the characters on the screen are kept) IN THE SAME CHARSET COMMAND, TO THE SAME 16K BLOCK AS THE CHARACTER SET. To do this, you will need a full definition of the CHARSET command:

CHARSET[cs][,sr]
(chA)

This command is used to move the position of the character set, and/or the position of the screen RAM.

cs, between 0 and 31, is used to specify the new position of the character set. This is found by dividing the address of the character set by \$800. e.g. to put the character set at \$7800, cs would be $\$7800/\800 , which is 15.

sr, between 0 and 63, is used to specify the new position of the screen RAM. It is found by dividing the address of the screen RAM by \$400.

NOTES

Although both parameters cs and sr are indicated as being optional, at least one must be present. If one is omitted, the current value of the other one is taken. THE CHARSET COMMAND MUST RESULT IN BOTH THE CHARACTER SET AND SCREEN RAM BEING IN THE SAME 16K BLOCK, or an error will result. So CHARSET 15,29 is fine, since:

$15 \times \$800 = \7800
and $29 \times \$400 = \7400 , and these are in the same 16K block.

But CHARSET 15,1 or CHARSET 15 while the screen RAM is not in the \$4000-\$7FFF block will both return errors.

(HINT: When dealing with user-defined graphics, or UDGs, you may find the '@' function useful - this converts decimal numbers to hex. So PRINT @1234 prints 4D2; PRINT @(15*\$800) prints 7800.)

NOW: to waste as little memory as possible, the screen RAM and character set should go to the top of user RAM. We shall put the screen RAM at \$7400 and the character set at \$7800-\$7FFF.

```
HIMEM=$7400:CHARSET 15,29:COPY 0 TO 255:PRINT"[cls]"
```

The HIMEM command is to protect the RAM against interference from BASIC; the CHARSET command moves the character set and screen RAM, as explained above; the COPY command copies the ROM characters into RAM; and then the screen is cleared to remove any initial garbage. This line is the standard first line in any UDG program.

The default (normal) CHARSET setting is CHARSET 2,1 (i.e. the character set is at \$1000, where the VIC chip sees the character ROM, and the screen RAM is at 1*\$400=\$0400). So, if you wish to revert to normal characters at the end of a program, use:

```
CHARSET 2,1:HIMEM=$8000:PRINT"[cls]":END
```

All these functions, apart from the HIMEM command, are performed by pressing RUN/STOP and RESTORE together.

Now you have a set of powerful user-defined graphic commands at your disposal; chapter 12 deals with more advanced character graphics, such as defining your own multi-colour characters.

Chapter 3

BASIC SPRITES

Movement using PRINT AT and user-defined characters is all very well, but it has its drawbacks. Firstly, movement tends to be jerky, as characters must be moved from one screen location to the next - i.e. 8 dots at a time. UDGs are small - only 8 by 8 dots, and making larger shapes by joining up smaller ones is tedious. Also, moving characters with PRINT AT tends to wipe away whatever was behind them (e.g. if you start with a starry background, the stars will gradually disappear as characters move over them).

As a solution to all these problems, the VIC chip enables you to use Sprites (or MOBs, Moveable Object Blocks, as Commodore prefers to call them). These are a powerful graphic feature practically unique to the Commodore 64. You can display up to 8 sprites at one time; each is defined on a large, 21 by 24 matrix of dots; they may be positioned anywhere on the screen, not being limited to 8 by 8 character squares and not interfering with screen data; they may be optionally expanded x2 and in the x-and/or y-directions; they have priorities, i.e. some go in front of others; they may be multi-coloured; and there is automatic collision-detection between sprites and the background and sprites and other sprites. Sprites offer a quick and easy way to create superb smooth movement and animation.

This chapter will deal with the basic Sprite facilities; advanced sprites, including interrupt-driven (automatically moving) sprites, will be dealt with in chapter 9.

a). Where do I put my sprites in memory?

A sprite definition is 64 bytes (actually 63 bytes plus 1 spare), and there are 256 such 'sprite blocks', numbered 0-255, in each 16K block of memory which the VIC chip can access. Fortunately, you do not have to reserve memory for sprites (although you can if you want to) because there is some memory free. Sprite block 11 is free, whatever happens. Sprite blocks 13 to 15 inclusive are also free, but these are in the cassette buffer, and are corrupted by tape operations such as LOAD and SAVE. Similarly to user-defined graphics, sprites must be defined on a 64-byte boundary, i.e. at \$0000,\$0040,\$0080 etc.

If you have moved VIC away from block 0, either by using CHARSET or because you are in a hi-res graphics mode, you will have to find other memory space for your sprites. This will be explained in chapter 9.

b). How do I put them there?

The process is similar to defining user-defined graphics, except you use the DEFSPR command.

Firstly, draw up a 21 by 24 matrix of squares, like that on page 70 of the Commodore 64 User Manual, and define your sprite on it. (N.B. The rows are numbered 0-20 in BC BASIC, not 1-21 as in the manual).

Now you can define your sprite:

```
DEFSPR 13,0,%000000000111111100000000
DEFSPR 13,1,%000000001111111111000000
DEFSPR 13,2,%000000011111111111100000
etc.
```

This defines the first 3 rows of the 'Commodore balloon' in the manual, into sprite block 13. Carry on until you have done row 20 - it is a long job, but worth it.

If you want to keep your sprite, put all these DEFSPR commands as program lines, and then run the program. You can then save the program later, or edit the sprite.

HINT: To clear a sprite, use the following line:

```
FOR R=0 TO 20:DEFSPR 13,R,0:NEXT R
```

In this DEFSPR command, 13 is the number of the block being cleared, R is a row counter variable, and 0 is the same as %000000000000000000000000.

A more efficient way of defining sprites will be dealt with later in this chapter.

c). Now how do I display it on the screen?

There are 8 sprites, numbered 0 to 7. These may all be used, but for the moment we will just use one of them. Choose one - let's say 0 for simplicity.

Firstly, you must switch it on - this is done by the SPRON command (all sprite commands in BC BASIC, apart from DEFSPR, begin with the three letters SPR).

```
SPRON 0,1
```

The first 0 is the number of the sprite. The 1 after the comma may be any non-zero value - it indicates that we want to switch the sprite ON. If it had been zero, the sprite would have been switched OFF.

Nothing happened, right? Well, that is because the sprite is positioned off the screen, where we can't see it. But first, we must tell the VIC chip which 'sprite block' it is to use - remember that there are 256 sprite blocks in each 16K VIC block of RAM.

What we are doing is setting the VIC sprite pointers - therefore we use the SPRPOINT command.

```
SPRPOINT 0,13
```


The 0 is the number of the sprite - 13 is the number of the block. If you defined your sprite in a different sprite block, say 11, then substitute this instead.

Now we are ready to move the sprite onto the screen. We are changing its position, so we use the SPRPOS command.

```
SPRPOS 0,100,100
```

Again, the 0 is the number of the sprite. The first 100 is the new x position of the sprite; the second 100 is its y position.

If all has gone well, your sprite should now be visible in all its glory.

If you don't like its colour, use the SPRINK command, INK signifying the colour the sprite is drawn in.

```
SPRINK 0,10
```

Yet again, the 0 is the sprite number. The 10, which may be any number in the range 0-15, is the new colour of the sprite - light red in this case.

Now try putting another sprite on the screen:

```
SPRON 5,1:SPRPOINT 5,13:SPRPOS 5,100,150:SPRINK 5,5
```

Another sprite, this time green and numbered 5, will appear a little way below the original sprite. Notice that it is getting its information from the same sprite block as sprite 0, i.e. block 13 - in fact, all the sprites may access the same block, or they may all be different.

Part of the power of BC BASIC sprite commands lies in the fact that many of them have several different formats (e.g. SPRON has three) so that you may control more than one sprite in each command.

Time for a few command definitions:

SPRCLR
(sprcL)

This command turns off all sprites, sets all positions to 0, clears the X and Y expand registers, and sets all the sprite INK and multi-colour colours to the current INK colour. THIS COMMAND SHOULD BE USED AT THE BEGINNING OF EVERY SPRITE PROGRAM to ensure that no spurious sprites are on, and that the sprites are not expanded etc.

SPRON n,e
or SPRON n TO n2,e
or SPRON=n

This command is used to enable or disable sprites. The first format you have already met - if e is non-zero, sprite n is enabled; if e is zero, sprite n is disabled. The second format does the same, but for all sprites from n to n2 inclusive. It does not

matter which way round n and n2 are.

e.g. SPRON 3 TO 6,1 switches on sprites 3,4,5 and 6.

The third format is used to set the entire 'sprite enable register' in VIC in one go, to n which is in the range 0-255.

e.g. (76543210)

SPRON=800111011

This switches on sprites 0,1,3,4 and 5, and switches off sprites 2,6 and 7.

SPRINK n[TO n2],col This command is used to set the colour of a sprite or sprites. If 'TO n2' is not included, then only sprite n is affected; if 'TO n2' is specified, then the colours of all sprites n to n2 inclusive will be set to 'col'.

SPRPOINT n[TO n2],p (sprpOI) Similar to SPRINK, this command sets the sprite pointers of one or more sprites to p, which must be between 0 and 255. If 'TO n2' is not included in the command, then only the pointer of sprite n is affected; if 'TO n2' is specified, all sprites from n to n2 inclusive will have their pointers set to p.

SPREXP n[TO n2],xe,ye
or **SPREXP=xe,ye (spreX)** This command is similar to SPRON. It sets the x and y expansions of one or more sprites. The first format may specify either one sprite (n) or several sprites (n TO n2 inclusive). If xe is non-zero, the specified sprite or sprites are doubled in width in the x direction; if xe is zero, they are set to normal width. The same applies to ye for the height of the sprites.

The second format sets the X and Y expand registers in the VIC chip in one go, like SPRON=. xe and ye must be between 0 and 255 - in binary, their bit patterns set which sprites will be normal (a 0 bit) and which will be expanded (a 1 bit).

SPRX n,xp This command sets the x-position of sprite n to xp. It performs two special functions which are normally very difficult in standard BASIC. Firstly, it deals with the ninth bit of the x-position automatically - so you can move right the way across the screen without the hassle involved with attending to the register for the top bit of the sprite positions. Secondly, it tells the difference between PAL and NTSC versions of the Commodore 64; if xp goes negative, the

computer decides whether the sprite should move to position 503 (PAL computers) or 511 (NTSC computers). xp should always be between -504 and +511. You will only need to use negative values if you are using double-width sprites.

SPRY n,yp

This command sets the y position of sprite n to yp, which is in the range 0-255. A y-position of 0 is off the top of the screen; if you increase yp, the sprite moves downwards.

SPRPOS n,xp,yp

A combination of the SPRX and SPRY commands, this command is a convenient way of setting the position of a sprite in one go. The same rules apply to xp and yp as if you were using the SPRX and SPRY commands separately.

A more efficient way of defining sprites

Using 21 DEFSPR commands in a program to define one sprite is both wasteful of memory and time-consuming. A better way is to use READ and DATA statements, as follows:

```
10 FOR K=0 TO 20:READ S$:DEFSPR 13,K,VAL("%"+S$):NEXTK
```

Continued....

```

9000 DATA 000000000000000000000000
9001 DATA 000000000000000000000000
.
.
.
9020 DATA 000000000000000000000000

```

You can create lines 9000-9020 by entering line 9000, and then moving up the cursor, incrementing the line number and then pressing RETURN until all 21 lines are present. Then LIST9000-9020 and you can move the cursor up and define your sprite by typing a 1 wherever you want the screen to be illuminated.

When you have finalised your sprite, you can compact the program further (to make it easier for anyone else typing it in) by converting the binary number in each data statement to either decimal or hex, and modifying line 10 slightly.

BC BASIC Sprite functions

BC BASIC gives you several functions to read sprite parameters - to see what colour a sprite is, whether it is on or off, or what its position on the screen is, for example. Here are some function definitions:

```

SPRDEF(n,rw)      This function returns the definition of
(sprdE)           sprite n, row rw (in the range 0-20) as a
                  number in the range 0 to $FFFFFF (which is
                  %111111111111111111111111).
                  e.g.      10 FOR R=0 TO 20
                           20 PRINT BIN$(SPRDEF(13,R),24,46,209)
                           30 NEXT R

```

This prints the definition of sprite 13 as a matrix of characters 24 across by 21 down.

```

SPRINK(n)          This function returns the colour of sprite n,
                  as a number in the range 0-15.

```

```

SPRPOINT(n)        This function returns the current value of
(sprpoI)           the sprite pointer for sprite n, in the range
                  0-255.

```

```

SPRX(n)            This function returns the X position of
                  sprite n as a number in the range 0-511.
                  Remember, for PAL computers, -1=503 and for
                  NTSC computers, -1=511. You only need to
                  remember this if you are using expanded
                  sprites. To see what sort of computer you
                  have, PRINT PEEK($2A6) - 0 is NTSC, 1 is PAL.

```

```

SPRY(n)            This function returns the Y position of
                  sprite n as a number between 0 and 255.

```

SPRON(n)
or SPRON

This function, which is used to read the sprite enable register, has two formats. The first, where it is followed by (n), returns 0 if sprite n is disabled and -1 if sprite n is enabled (-1 = logical 'TRUE'). The second format merely reads the entire sprite enable register, and returns a value between 0 and 255, where each bit represents whether that particular sprite is on or off.

SPREXPX(n)
or SPREXPX
(spreXx)

This function is to check if a sprite is expanded in the x direction or not. The first format, followed by (n), checks sprite n, returning 0 if it is unexpanded or -1 if it is expanded in the x direction. The second format reads the entire sprite x-expand register, returning a value between 0 and 255.

SPREXPY(n)
or SPREXPY
(spreXy)

As for SPREXPX, but for expansion in the y direction rather than the x direction.

Sprite Priorities

This is very simple. When two or more sprites are on top of each other, the sprite with the lowest number will be displayed - so sprite 0 is displayed on top of all the other sprites, sprite 1 is on top of sprites 2-7 but beneath sprite 0, and sprite 7 is always bottom.

A demonstration sprite program

```
10 SPRCLR:SPRPOINT 0TO7,11:SPRON 0TO7,1:PRINT"[cls]"
20 FOR K=0 TO 7:SPRINK K,K+2:NEXT K
30 FOR K=0 TO 20:READ A$:DEFSPR 11,K,VAL("%"+A$):NEXTK
40 FOR S=1 TO 20
50 FOR K=0 TO 7:SPRPOS K,30+S*K,120+K*2:NEXT K
60 NEXT S
70 FOR K=0 TO 7
80 FOR Y=SPRY(K)TO0STEP-2:SPRY K,Y:NEXT Y
90 NEXT K:SPRON=0:END
100 DATA 00000000000000000000000000000000
101 DATA 00000000000000000000000000000000

. define your own sprite in lines 100-120 inclusive

120 DATA 00000000000000000000000000000000
```

Lines 10-30 initialise the sprites; lines 40-60 make them spread out across the screen; lines 70-90 make them move up and off the screen, one by one. The 21 lines from 100-120 inclusive are for you to define your sprite.

Try adding

15 SPREXP 0 TO 7,1,1 to enlarge the
sprites.

A VERY IMPORTANT POINT

By now, you will be writing sprite programs worth saving. However, since having sprites on the screen slows down the processor chip slightly, this creates errors with important cassette and disk timings. Therefore, ALWAYS switch off all sprites (the best way is to type SPRCLR) before any cassette or disk load or save operations, or ?LOAD ERRORS may result.

Chapter 4

BASIC INPUT/OUTPUT

To enable you to make full use of the sprite and user-defined graphics facilities you have already learnt, this chapter deals with reading the keyboard and joysticks. More advanced I/O, namely using the user-port, defining the functions keys and scanning individual keys on the keyboard directly, are covered in chapter 11.

Firstly, here are two functions to read which key is being pressed.

KEY
(kE)
This function returns the ASCII code of the key which is currently being pressed, or 0 if no key is pressed. Unlike GET, KEY repeats if a key is held.

KEY\$
(kE\$)
This function returns the key which is currently being pressed as a string, or a null string ("") if no key is pressed. Like KEY, this function also repeats if the key is held.

e.g. this could be an extract from a game program:

```
50 IF KEY$="A" THEN X=X-1:REM MOVE LEFT
60 IF KEY$="S" THEN X=X+1:REM MOVE RIGHT
70 IF KEY=13 THEN GOTO 200:REM FIRE!
```

In line 70, 13 is the ASCII code for RETURN - i.e. the return key is the fire button.

Now for three powerful functions to read the joysticks.

JOY(n)
or **JOY(n,b)**
(jO)
This function is used to read joystick n, where n is 1 or 2. If the format JOY(n) is used, the number returned is in the range 0-31. In binary, bit 0 (the right-hand one) is UP, bit 1 is DOWN, bit 2 is LEFT, bit 3 is RIGHT and bit 4 is FIRE. Each bit is 0 if the switch is open, 1 if it is closed. The second format allows you to specify which bit to read - b is in the range 0-4, as above. 0 is returned if the switch is open; -1 (logical 'TRUE') if it is closed. e.g. JOY(2,4) returns 0 if the fire button of joystick 2 is not pressed, -1 if it is.

JOYX(n)
(jOx)

This function returns the x-movement in joystick n; i.e. -1 if it is pulled to the left, 0 if it is centred and 1 if it is pulled to the right. It is independent of movement in the y (up and down) direction.

JOYY(n)
(jOy)

As for JOYX, but returns movement in the y-direction; i.e. -1 if the stick is pulled up, 0 if it is centred in the y-direction, and 1 if it is pulled down.

Together, the JOYX and JOYY functions make movement a cinch!

```
e.g. 10 X=100:Y=100
      20 SPRPOS 0,X,Y
      30 X=X+JOYX(1):Y=Y+JOYY(1):GOTO 20
```

This is all the code that is required to move a sprite around the screen under joystick control - even diagonally.

Chapter 5

A FIRST LOOK AT STRUCTURED PROGRAMMING

Take the following line of BASIC:

```
100 GOSUB 13500
```

On its own, it is meaningless - it gives no indication of what function the subroutine at 13500 performs. The following is little better:

```
100 GOSUB 13500:REM DISPLAY MENU
```

since it is clumsy, and requires you to remember when writing your program that the 'Display Menu' subroutine is at line 13500.

Another problem is that the subroutine at line 13500 may use variables which are already in use, disrupting their contents by the time it returns. Clearly this is completely unacceptable!

BC BASIC overcomes these limitations by means of neat programming structures known as procedures. They are being introduced here as you will find them useful in the next chapter, which deals with high-resolution graphics.

In their simplest form, procedures are named subroutines. For example:

```
13500 DEFPROC MENU
13510 REM DISPLAY MENU, HERE WE GO...
.
. program code
.
13800 ENDPROC
```

In line 13500, DEPROC (DEFine PROCedure) indicates the start of the procedure called 'MENU'. In line 13800, ENDPROC is the equivalent of RETURN - it returns to whatever line the procedure was called from. It is called by:

```
100 PROC MENU
    (prO)
```

A few notes at this stage:

There are few limitations to procedure names. They must not contain graphic symbols, double-quotes (shift and 2) or left-hand brackets. Apart from this, they may be of any length, and may even contain keywords like PRINT and LET. DEF and PROC must be the first two words on a line (although spaces may be inserted between DEF and PROC). Spaces between PROC and the beginning of the procedure name are irrelevant, but spaces actually in the procedure name must match exactly. Perhaps most importantly, YOU

CANNOT EXECUTE A 'DEPROC' - it will give a ?SYNTAX ERROR if you try to. Instead, the words DEF and PROC are searched for when you call a procedure. It is good practice to keep all procedures at the end of a program.

The next important facility of procedures is the ability to use local variables - in effect, variables which can only be 'seen' inside the procedure itself. Upon returning, the values of local variables are restored. The following command is used to define local variables:

```
LOCAL variable(s)
(loC)
```

If there is more than one variable, separate them by commas. This command remembers the old contents of variables, so they will be restored by the ENDPROC command. Variables may be normal, integer, string, or elements of an array.

e.g. LOCAL A,F%,C\$,A(3),B%(9,7),S\$(2)

It should be obvious that LOCAL may only be used within a procedure.

An even more powerful feature of BC BASIC procedures is that parameters, or numbers to be used, may be passed in brackets. For example:

```
13500 DEFPROC MENU (A,B)
```

If this is called by PROC MENU (3,5) then A is set to 3 and B set to 5 - PLUS A and B are automatically defined as LOCAL, so that if they were being used before, their old values will be restored. This facility is extremely powerful - even strings may be passed to procedures.

A final point: like GOSUBs, PROCedures may call other procedures, still with all the LOCAL variable facilities.

Here is an example of using procedures:

```
Type:      10 DEFPROC TEST(A$,B,C):LOCAL D
            20 PRINT"A$=";A$:PRINT"B=";B:PRINT"C=";C
            30 D=B+C:PRINT"D=B+C=";D:ENDPROC
```

```
Now type:  A$="ABC":B=39:C=87:D=22
            PROC TEST("HELLO",3,7)
            PRINT A$,B,C,D
```

You should find that A\$,B,C, and D have kept their original values, even though they are used in the procedure.

IMPORTANT POINT: If you wish to make sure that your procedures are completely self-contained, then do not use FOR...NEXT loops inside them. The reasons for this are complex, but are to do

with the fact that 64 BASIC keeps a separate record of FOR...NEXT loops (as well as the value of the variable) and if you terminate such a loop within a procedure, BASIC thinks the same loop is terminated outside the procedure (if one was in use with the same variable).

You should try to get into the habit of breaking your program into chunks, and then defining them as PROCedures - it makes program writing and debugging much faster and easier.

Chapter 6

HIGH-RESOLUTION GRAPHICS

We now come to a very important aspect of the Commodore 64 - bit-mapped graphics, or high-resolution graphics, or just hi-res for short. It is possibly one of the main reasons why you bought the BC BASIC cartridge.

Bit-mapping, a very popular computer graphics technique, allows you to control each individual DOT on the screen, independant of the others; to draw lines of any length and direction; and to create very detailed pictures, graphs and diagrams.

BC BASIC has two hi-res graphics modes, MODE 3 and 4, which will be explained in detail shortly - however, briefly, MODE 3 allows a resolution of 200 dots down the screen by 320 dots across it, with a foreground and background colour selectable for each 8 by 8 dot square; and MODE 4 allows a resolution of 200 dots down by 160 dots across, but each square may contain dots in three different colours, plus the background, which is the same for the whole screen.

You have two hi-res screens to use, numbered 0 and 1, which are totally independent of each other - so you may be displaying one of them while you are preparing the other one to be displayed. You may also put text on the hi-res screens using the HPRINT command, and this text may be single or double-width. In MODE 3, HPRINT has the advantage over normal text mode that each character on the screen may have its own PAPER colour as well as its own INK colour, but it is slightly slower than PRINT (nevertheless, it is fast enough for most applications).

The two hi-res screens are separate from the text screen - you must display either one or the other. Therefore, you cannot see what you are doing in hi-res mode, but programs may output to the screen using HPRINT, e.g. for labelling graphs.

Telling the computer to use hi-res mode

The command MODE is used to change graphics modes. For the moment we shall concentrate on MODE 3, the STANDARD BIT-MAP MODE, but we shall look at MODE 4 later.

To switch into hi-res mode, type the following:

```
MODE 3,0:CLG
(mO)
```

The 3 tells the computer which mode you wish to enter - in this case, standard hi-res. The 0 after the comma is the number of the screen you wish to use, which is 0 or 1. The CLG for Clear Graphics, clears the screen of anything which was on it beforehand. You don't have to use a CLG after a MODE command - we just want a clear screen so we can start experimenting.

You will now have a blank screen in front of you, with no cursor. Anything you type will not be displayed - this is because it is being written to the text screen, which is independent of the hi-res screen you are using.

To return to standard text mode, type the following:

MODE 0

The text screen will be unchanged, just as you left it before going into hi-res mode, unless you have typed anything else in the mean time. MODEs 1 and 2 are also possible - these are different character modes. These will be explained in chapter 12.

Note that the commands MODE 0, MODE 1 and MODE 2 must not have a number following them, but MODE 3 or MODE 4 must be followed by a comma and then 0 or 1 to specify which graphics screen is to be displayed.

HINT: In some programs, it will be necessary to switch rapidly from mode to mode, or between the two graphics screens. In this case, to prevent 'screen access noise' (flicker) on your TV, precede the MODE command directly by a SCRWAIT command. e.g.

```
10 SCRWAIT:MODE 3,0
20 SCRWAIT:MODE 3,1
30 GOTO 10
```

This little program switches very quickly between the two hi-res screens, but cleanly, because SCRWAIT waits until the screen is not being displayed (i.e. the border at the top or bottom of the screen is being displayed). SCRWAIT, which is abbreviated to sC, may also be used with other graphics (e.g. PRINT AT) to reduce flicker.

Plotting points on the hi-res screen

Now you know how to switch to MODE 3. However the resolution, 200 by 320 dots, is very high for a colour TV, and, if you are not careful, you will get 'chroma noise' (colour distortion) displayed. You must either make all dots and lines at least 2 dots wide (just like when programming UDGs) or you must choose your colours very carefully. Experiment with them - a good combination is light grey (15) on a green background (5).

YOU SHOULD SET YOUR COLOURS, USING THE 'PAPER' AND 'INK' COMMANDS, BEFORE YOU DO A 'CLG'. This is because CLG takes the values written in the INK and PAPER registers, and writes them to the hi-res colour RAM, as well as clearing the screen.

You may change the INK as often as you like when plotting in hi-res - but be careful if you come into an 8 by 8 character square where other points have been plotted - ALL the points in that 8 by 8 square will be set to the current INK colour, and this can

lead to a messy display.

The PAPER value has no effect, except when you perform a CLG or HPRINT operation.

To plot a point you use the PLOT comand (abbreviated to pL). This is followed by two numbers, separated by a comma: the first is the x coordinate of the point being plotted, and is in the range 0-319. The second is the y coordinate, which is 0-199. Numbers in the range -32768 to 32767 may be used without generating errors, but points outside the screen range will not be plotted. Note that the point 0,0 is the bottom left-hand corner of the screen, unlike PRINT AT.

Try the following program:

```
10 MODE 3,0:REM SELECT HI-RES, SCREEN 0.
20 PAPER 5:INK 15:REM SET COLOURS
30 CLG:REM CLEAR HI-RES SCREEN
40 FOR X=0 TO 319
50 PLOT X,SIN(X/50)*90+100
60 NEXT X
```

A 'sine curve' will be displayed on the screen as a series of dots. The operation is slow, but this is because the SIN function takes a long time.

When the program stops, typing MODE 0 will return you to text mode. To change the colours back, type INK 3:PAPER 11. Alternatively, pressing RUN/STOP and RESTORE together will perform both these functions.

Unplotting and inverting points

Putting a point onto the screen is fine - but you may want to take a point off, or 'invert' it - invert means switch it on if it is off, and switch if off it it is on. These functions are achieved by the addition of a comma plus a number onto the end of the PLOT statement:

PLOT X,Y,0	unplots the point X,Y (turns it off)
PLOT X,Y,1	plots the point X,Y (turns it on)
PLOT X,Y,13	inverts the point X,Y (changes it)
PLOT X,Y,8	does nothing *

* used with DRAW to specify where to draw from.

The command GCOL (abbreviation gC) is used to set the default 'logical plotting colour' or method of plotting: the default is the one which will be used if no number follows the plot statement.

e.g. GCOL 13
 PLOT X,Y

is the same as PLOT X,Y,13, except that ALL PLOTs and DRAWs from now on will be plotted '13'. The GCOL is automatically set to

1 by CLG - that is why PLOT normally turns a point on if no number follows it.

Testing points

Sometimes it is necessary to see if a point is set or not. This is tested by the POINT function. It is followed by an X,Y coordinate in brackets.

e.g. A=POINT(100,150) will set A to 0 if the point is not set or 1 if it is set.
 A=POINT (100,250) will set A to -1, since the point is off the screen.

Drawing lines

This is performed by the DRAW command. It can be used to either draw a line from the last point PLOTted or DRAWn, or it can be used to draw a line between two specified points.

e.g. DRAW X1,Y1 TO X2,Y2 draws a line from X1,Y1 to (dr) X2,Y2 inclusive.
 DRAW TO X1,Y1 draws a line from the last point plotted or drawn to X1,Y1.

DRAW is affected by GCOL just like PLOT, and may also be followed by a comma and GCOL number: so,

DRAW 50, 50 TO 300,150,13

inverts all the points in a straight line from 50,50 to 300,150 inclusive.

PLOT X,Y,8:DRAW TO X2,Y2 is the same as DRAW X,Y TO X2,Y2.

If EITHER of the points are off the screen, NO LINE IS DRAWN.

So, the sine-wave program on the previous page can be modified as follows:

```
35 PLOT 0,100
40 FOR X=5 TO 315 STEP 5
50 DRAW TO X,SIN(X/50)*90+100
```

These modifications draw a smooth curve, but very quickly, because the STEP 5 in line 40 means that 5 times less SINES have to be calculated by the computer.

NOTE: CLG sets the 'last point plotted' register to (0,0), so DRAW TO will draw from 0,0.

Drawing circles and rectangles

These are most conveniently achieved by using procedures - put them at the end of your program. There would have been little point in actually building a CIRCLE command into BC BASIC, since its speed is limited by the SIN function (so BASIC is just as fast as machine-code), and using a procedure means that you can modify the command (e.g. to draw arcs or ellipses).

```
60000 DEFPROC CIRCLE(X,Y,R):LOCAL A
60010 PLOT X+R,Y,8:A=.2
60020 REPEAT DRAW TO X+COS(A)*R,Y+SIN(A)*R
60030 A=A+.2:UNTIL A>6.4:ENDPROC
60100 DEFPROC REC(X1,Y1,X2,Y2)
60110 DRAW X1,Y1 TO X2,Y1
60120 DRAW X1,Y1 TO X1,Y2
60130 DRAW X2,Y1 TO X2,Y2
60140 DRAW X1,Y2 TO X2,Y2
60150 ENDPROC
```

as examples of these procedures, PROC CIRCLE(100,150,20) draws a circle of centre 100,150 and radius 20; and PROC REC(10,20,200,150) draws a rectangle which has diagonally opposite corners at 10,20 and 200,150. The REPEAT...UNTIL structure in lines 60020-60030 will be new to you - it says 'keep doing this until A becomes larger than 6.4'. It could have been replaced by:

```
60020 DRAW TO X+COS(A)*R,Y+SIN(A)*R
60030 A=A+.2:IF A<=6.4 THEN 60020
60040 ENDPROC
```

Remember not to use FOR...NEXT loops in procedures, as these may affect FOR...NEXT loops in operations outside of the procedure.

ST - testing if a point is off the screen

After every PLOT, DRAW or POINT operation, BC BASIC sets the ST system variable to 0 if the operation was carried out successfully, or -1 if the point (or one of the points) was off the screen. This means that a 'point off screen' status can be checked for without several IF statements.

ST will only give a valid answer if the current I/O device is NOT 2 - i.e. if the RS232 port is not being used. If an RS232 device is in use, use PEEK(\$90) instead of ST.

HPRINT - printing text onto hi-res screens

The format of this command is as follows:

```
HPRINT AT y,x;expression(s)
(hP)
```

y,x is exactly the same location as you would use in PRINT AT -

i.e. y is between 0 and 24, x is 0-39 and 0,0 is the top-left corner. Each expression may be a string expression or a numeric expression - unlike PRINT, numbers are displayed without a trailing (following) space. The expressions may be separated by semicolons, commas or nothing at all, but all have the same effect - i.e. unlike PRINT, ' ' = ' ';'. However, there must be no comma or semicolon at the end of the line.

e.g. HPRINT AT 3,0;"ANSWER IS",A+B"(=A+B)"
 prints ANSWER IS xxx(=A+B) on the hi-res screen at 3,0,
 where xxx is the result of A+B.

An expression may also consist of a hash ('#', shift-3) followed by a number or numeric expression giving 0 or 1. If it is 0, printing is set to normal size; if it is 1, printing from then on until the next #0 or the end of the command is in double-width lettering (e.g. for bold titles)

e.g. HPRINT AT 5,5;#A;"ABC"

prints ABC in normal width if A=0 or double width if A=1.

When characters are displayed on the hi-res screen using HPRINT, the INK and PAPER values of each character square are set to the current INK and PAPER values. Therefore, HPRINT may be used to set areas of the screen to a new PAPER (by changing the PAPER then HPRINTING spaces), and each character on the screen may have its own foreground and background colour.

HPRINT ignores control characters in strings - if reverse is required, swap the INK and PAPER values.

If text goes off the screen using HPRINT, it has no effect - i.e. the screen is not scrolled.

Finally, HPRINT does respond to the CHARSET setting (it calls the CHARSET function to get its data) and so user-defined characters may be printed to the hi-res graphics screens.

A note about CHARSET in hi-res

CHARSET, the command for user-defined graphics, actually has a different format in hi-res mode. Firstly, since it is not the VIC chip which is reading the character set (it is BC BASIC, which then writes it to the hi-res screen) there are no limitations on 16K blocks - CHARSET may be followed by any number in the range 0-31. Secondly, it is meaningless to try to move the screen RAM while in hi-res mode, so CHARSET may not be followed by a comma and the new address of screen RAM.

The RAM behind the BC BASIC ROM at \$8000-\$8BFF is free, so an ideal position for hi-res UDGs is at \$8000-\$87FF (CHARSET 16), which uses no user RAM. If you want to use your UDGs in both hi-res and normal modes, then use CHARSET 15 as before (use CHARSET 15,29 in normal mode and HIMEM=\$7400 at the beginning of the program).

NOTE that selecting MODEs 0,1 or 2 always resets the CHARSET to 2

(the default value) so a CHARSET command may be needed after each MODE command of this sort.

Here is a short program using HPRINT:

```
10 PAPER 6:INK 14:MODE 3,0:CLG
20 CHARSET 16:COPY 0 TO 255
30 C(0)=10:C(1)=14:C(2)=15
40 DEFUSR 42,%00011000,%00111100,%01111110,%1101101
50 DEFUSR 42;4;%11111111,%11000011,%01111110,%11000011
60 PAPER 5:INK 15
70 HPRINT AT 1,0;#1;" BC BASIC ";#0;"FOR COMMODORE 64 "
80 PAPER 6:FOR K=1 TO 50
90 INK C(RND(1)*3)
100 HPRINT AT RND(1)*22+3,RND(1)*40;#RND(1)*2;"*"
110 NEXTK
120 PAPER 11:INK 3:FOR K=1 TO 2000:NEXT:MODE0:END
```

Lines 10-50 set up graphics; lines 60-70 print a title; line 80 starts a loop; line 90 picks a random colour; line 100 prints a star (actually a UDG) at a random position with a random enlargement; line 110 loops round, and line 120 goes back to normal mode after a short delay.

Specifying the plotting screen

MODE can be used to determine which screen is to be plotted to, but it also makes that screen be displayed. When you need to plot to one screen whilst displaying the other, the command PMODE (plotting mode) is used - it sets which screen is to be plotted to, without changing the graphics mode.

Format PMODE m,s
 (pM)

m is to specify whether the screen is to be displayed in MODE 3 or MODE 4. If the screen is to be displayed in MODE 3, m should be 0 - if it is to be displayed in MODE 4, m should be 1. This parameter is important, as BC BASIC must be told whether it is to plot its points as if it were in MODE 3 or MODE 4.

s is the parameter we are interested in - again 0 or 1, this number specifies which screen is to be plotted to.

REMEMBER that m and s are reset by MODE, so PMODE should come AFTER MODE.

This is a common statement with PMODE:

```
SCRWAIT:MODE 3,S:S=1-S:PMODE 0,S:CLG
```

All PLOT, DRAW and HPRINT operations after this will go to the opposite hi-res screen to the one which is being displayed. The next time this line is executed, the screens will be swapped again. S holds the number of the current plotting screen.

```
e.g.      10 FOR A=0 TO 2* $\pi$  STEP .1
          20 SCRWAIT:MODE 3,S:S=1-S:PMODE 0,S:CLG
          30 DRAW 160,100 TO COS(A)*50+160,SIN(A)*50+100
          40 NEXT A:GOTO 10
```

Try this program - it moves a line around in a circle. Now try the following modification:

```
20 SCRWAIT:MODE 3,S:CLG
```

This is what normally happens with computers with only one graphics screen - the line flashes annoyingly.

MODE 4 - Multi-colour bit-map mode

This mode allows 3 colours, plus the background, in each 8 by 8 square. It is used:

- a). For graphs, when lines of different colours need to be able to cross.
- b). For extra colour clarity (because each dot displayed is actually 2 dots wide).

MODE 4 resolution is 200 dots down the screen by 160 dots across - however, the PLOT and DRAW and POINT coordinates are the same as for MODE 3: i.e. the x coordinate is still 0-319. This is to allow easy changing of programs between modes.

MODE 4 is slightly less flexible than MODE 3 because the background colour is the same for the whole screen - it is selected by the PAPER command. Also please note that, because of the lowered resolution, text should be HPRINTed in double-width only, or the characters will appear messy. Use SETCOL 3, col instead of INK col to set the text colour. This is because bits are taken in pairs in mode 4, and 11 in binary = 3 in decimal.

MODE 4 is selected by the command MODE 4,n where n is the number of the graphics screen being displayed. The four colours which will be displayed are selected as follows:

PAPER col	or SETCOL 0,col	sets the background colour (colour 0)
INK col	or SETCOL 1,col	sets colour 1
	SETCOL 2,col	sets colour 2
	SETCOL 3,col	sets colour 3

NOTE that MODE 4 uses the colour data RAM - therefore, when you switch back to MODE 0,1 or 2, the colours of your characters will be different to what you originally set them to.

Which of the 4 colours will actually be displayed is determined by the GCOL number, selected either by GCOL or by following the PLOT or DRAW command with a comma and a number.

Simply,

```

      ,0' unplots
      ,1' plots in colour 1
      ,2' plots in colour 2
and   ,3' plots in colour 3

```

However, the GCOL number may actually be between 0 and 15, and is very much more powerful than this. Here is a full table of GCOL values and their effects:

GCOL VALUE		EFFECT	
DECIMAL	BINARY	IN MODE 3	IN MODE 4
0	%0000	Unplots	Plots a '0' point - i.e. unplots:
1	%0001	Plots	Plots a '1' point (INK n)
2	%0010	As 0	Plots a '2' point (SETCOL 2,n)
3	%0011	As 1	Plots a '3' point (SETCOL 3,n)
4	%0100	Unplots	ANDs with '00' - i.e. unplots
5	%0101	Nothing	ANDs with '01' (0→0, 1→1, 2→0, 3→1)
6	%0110	As 4	ANDs with '10' (0→0, 1→0, 2→2, 3→2)
7	%0111	As 5	ANDs with '11' - i.e. nothing
8	%1000	Nothing	ORs with '00' - i.e. nothing
9	%1001	Plots	ORs with '01' (0→1, 1→1, 2→3, 3→3)
10	%1010	As 8	ORs with '10' (0→2, 1→3, 2→2, 3→3)
11	%1011	As 9	ORs with '11' - i.e. plots a '3' point
12	%1100	Nothing	XORs with '00' - i.e. nothing
13	%1101	Inverts	XORs with '01' (0→1, 1→0, 2→3, 3→2)
14	%1110	As 12	XORs with '10' (0→2, 1→3, 2→0, 3→1)
15	%1111	As 13	XORs with '11' (0→3, 1→2, 2→1, 3→0)

A series of numbers in brackets indicates the effect of the operation on each different type of point; e.g. (0→0, 1→1, 2→0, 3→1) means that a '0' point stays the same, as does a '1' point; a '2' point becomes a '0' point and a '3' point becomes a '1' point.

In MODE 4 the POINT function returns a number between 0 and 3, or -1 if the point specified is off the screen.

An example of the use of GCOL:

When drawing stereoscopic pictures, it is necessary to have a white background, to plot in red and green, and to have black where the lines cross. This is accomplished as follows.

The background, colour 0, is white

PAPER 15 (or SETCOL 0,15)

colour 1 is light red
colour 2 is green
colour 3 is black

INK 10 (or SETCOL 1,10)
SETCOL 2,5
SETCOL 3,0

When you wish to plot in green, use GCOL 10. As you can see from the table on the previous page, this converts 0 (white) to 2 (green), 1 (red) to 3 (black) and leaves 2 and 3 (green and black) unaffected.

When you wish to plot in red, use GCOL 9. From the table, this converts 0 (white) to 1 (red), 2 (green) to 3 (black) and leaves 1 and 3 (red and black) alone.

As you can see, the desired effect has been achieved simply and efficiently, with BC BASIC doing all the work.

Reading the hi-res colours

As each 8 by 8 character square may contain different colours, it is useful to be able to read which colours are in use in each square. This is achieved by the SCR and GCOL functions. You have used these functions before, and will know that the numbers in brackets range from (0,0) to (24,39), with (0,0) being the top left-hand corner. Clearly conversion from hi-res coordinates to screen coordinates is required, which is done as follows:

$X = \text{INT}(X/8) : Y = \text{INT}((199 - Y)/8)$

Now that X and Y have been converted to screen coordinates, you may read the colours.

MODE 3

The PAPER colour in each square is given by:

$(\text{SCR}(Y,X) \text{AND} 15)$

The INK colour in each square is given by:

$\text{INT}(\text{SCR}(Y,X)/16)$

MODE 4

Colour 1 in each square is given by:

$\text{INT}(\text{SCR}(Y,X)/16)$

Colour 2 in each square is given by:

$(\text{SCR}(Y,X) \text{AND} 15)$

Colour 3 in each square is given by:

$\text{GCOL}(Y,X)$

REMEMBER to convert hi-res coordinates (0-319,0-199) to PRINT AT

coordinates (0-24,0-39) before using these functions!

For the reference of advanced programmers

So as not to waste BASIC RAM, BC BASIC keeps the hi-res screen data behind the ROMs. Here is a table of the RAM addresses used by BC BASIC hi-res.

:-----:	:-----:	:-----:
:	: Bit memory	: Colour memory
:-----:	:-----:	:-----:
: Screen 0	: \$A000-\$BF3F	: \$8C00-\$8FE7
:-----:	:-----:	:-----:
: Screen 1	: \$E000-\$FF3F	: \$D000-\$D3E7
:-----:	:-----:	:-----:

A MODE 4 hi-res demo program

This program draws random lines in three different colours.

```
10 MODE 4,0
20 INK 5:SETCOL 2,10:SETCOL 3,15:CLG
30 FOR C=1 TO 3
40 DRAW RND(1)*320,RND(1)*200 TO RND(1)*320,RND
   (1)*200,C
45 REM the "C" selects which colour is to be
   used (1, 2 or 3)
50 NEXT C:GOTO 30
```

The program should be self-explanatory.


```

:-----:-----:
: 3      : Selects the PULSE waveform output. The harmonic:
:         : content of this waveform can be adjusted by PWM,:
:         : producing tone qualities ranging from a bright,:
:         : hollow square wave (2048) to a nasal, reedy pulse:
:         : (100-300). Sweeping the pulse width in real-time:
:         : produces a dynamic "phasing" effect which adds a:
:         : sense of motion to the sound. Rapidly jumping:
:         : between different pulse widths can produce:
:         : interesting harmonic sequences. PWM must be used:
:         : before this output becomes audible. (see later):
:-----:-----:
: 4      : Selects the NOISE output waveform. This output:
:         : is a random signal which changes at the frequency:
:         : of oscillator "V". The sound quality can be:
:         : varied from a low rumbling to hissing white noise:
:         : via the PITCH setting of the oscillator. Noise:
:         : is useful in creating explosions, gunshots, jet:
:         : engines, wind and other unpitched sounds, as well:
:         : as snare drums and cymbals. Sweeping the PITCH:
:         : with Noise selected produces a dramatic rushing:
:         : effect.
:-----:-----:

```

Note that an oscillator is the circuit which produced sounds; a channel or voice is the output from the oscillator, after having been subjected to the volume changes of the ADSR envelope (to be explained shortly) and other functions such as 'ring modulation' and 'synchronisation'.

Now type:

WAVEFORM 1,2

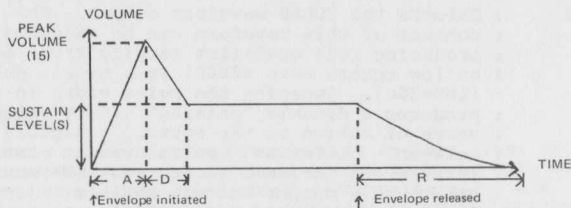
This selects the sawtooth waveform for channel 1. If you had selected waveform 3 (pulse), you would now have to type afterwards:

PWM 1,2048

PWM (for Pulse Width Modulate) specifies the pulse width of a pulse wave, which must be set before waveform 3 can be heard. As mentioned in the table of waveforms, 2048 gives a bright, hollow square wave; a value of about 300 is quieter and more reedy.

iv). Set the envelope, using the ADSR command - this is perhaps the trickiest stage of the operation. In everyday life, sounds do not start suddenly then stop suddenly - for example, a note played on a piano starts quickly (it has a fast attack) but then decays slowly. A xylophone has a much faster decay rate - the sound dies away more quickly. Real 'envelopes' (how the volume changes with time) are extremely complex, but synthesisers use a compromise technique called Attack-Decay-Sustain-Release, or ADSR. A graph of volume

against time would look like this:



The volume starts at 0. When the envelope is initiated, the volume rises to a maximum, taking time A. It then drops to the sustain level S, at a rate D (D is in fact the time it would take to drop all the way to zero). The volume stays at level S indefinitely, until the envelope is released, when the volume dies away to zero at rate R.

In BC BASIC, all parameters A,D,S and R are between 0 and 15. They are set by the ADSR command:

```
ADSR v,a,d,s,r
(aD)
```

s is the actual volume, between 0 and 15, at which the voice sustains - the other parameters a,d and r relate to real times as shown in the following table. Note that the DECAY/RELEASE rates refer to the amount of time these cycles would take to fall from maximum volume to zero.

VALUE	ATTACK TIME	DECAY/RELEASE TIME
0	2 ms	6 ms
1	8 ms	24 ms
2	16 ms	48 ms
3	24 ms	72 ms
4	38 ms	114 ms
5	56 ms	168 ms
6	68 ms	204 ms
7	80 ms	240 ms
8	100 ms	300 ms
9	250 ms	750 ms
10	500 ms	1.5 s
11	800 ms	2.4 s
12	1 s	3 s
13	3 s	9 s
14	5 s	15 s
15	8 s	24 s

(1 ms = $\frac{1}{1000}$ of a second)

Just for the moment, type the following:

```
ADSR 1,12,12,5,9
```

- v) Set the pitch of the note to be heard, using the PITCH command:

```
PITCH v,n  
(pI)
```

V is the voice (1-3); n is the pitch of the note, between 0 and 65535. It will be explained shortly how to calculate n for different notes, but for now type:

```
PITCH 1,5000
```

- vi) Now all that needs to be done is to initiate the ADSR envelope, by typing:

```
ATTACK 1  
(atT)
```

You should hear a buzz grow louder (attack) and die away to a quieter volume (decay to the sustain level). The Release phase of the cycle may now be started:

```
RELEASE 1  
(reL)
```

The volume will die away to practically nothing. Slow values of Attack, Decay and Release have been deliberately chosen so that you can hear the effect.

An investigation into envelopes and waveforms using BC BASIC

The 64 can actually act as an oscilloscope onto itself! The following two functions mean that you can see the status of the envelope and oscillator on channel 3.

ENV3 This function returns a number between 0 and 255 which is proportional to the volume of the envelope on channel 3. 0=minimum volume, 255=maximum volume. The following program demonstrates this effectively, by drawing a graph of the envelope:

```
10 ADSR 3,11,12,5,9:MODE 3,0  
20 CLG:ATTACK 3  
30 FOR J=1 TO 100:DRAW TO J*2,ENV3*.75:NEXT J  
40 RELEASE 3  
50 FOR J=101 TO 150:DRAW TO J*2,ENV3*.75:NEXT J  
60 GOTO 20
```

This program clearly shows the shape of the envelope. It should be fairly self-explanatory - it initialises then starts the envelope on channel 3, plots a graph,

and then when it has got a certain way across, it releases the envelope. Try altering the values in line 10 to see their effect. ENV3*.75 is used instead of just ENV3 because the y-coordinate in hi-res plotting can only go up to 199, and ENV3 goes up to 255.

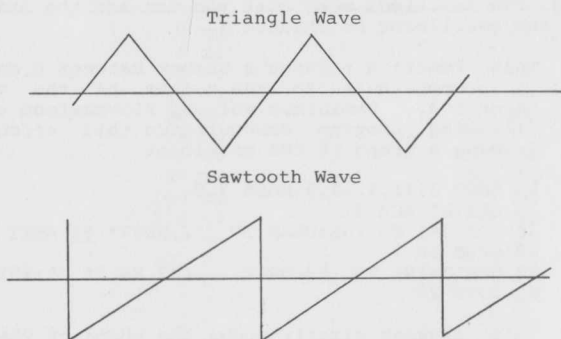
CH3 This function returns a value between 0 and 255 which is proportional to the voltage output of oscillator 3 (independent of envelope settings or VOLUME). It means that oscillator 3 can be monitored to modulate other oscillators - for example, a triangle waveform on channel 3 could produce a rising and falling siren on channel 1 or 2. Oscillator 3 set to noise output (WAVEFORM 3,4) will generate random numbers in CH3.

e.g. 10 SOUNDOFF:VOLUME 15
20 ADSR 1,0,0,15,0:ATTACK 1:WAVEFORM 1,2
30 WAVEFORM 3,2:PITCH 3,25
40 PITCH 1,CH3*10+5000:GOTO 40

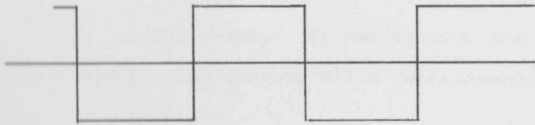
This plays a 'red-alert!' siren. Line 10 is basic initialisation; line 20 sets up voice 1, which will be used for actually making the sound (envelope settings of 0,0,15,0 make the voice turn straight on and off). Line 30 sets the waveform of channel 3 to a sawtooth, with a low frequency (pitch of 25). Line 40 then scales the value from CH3 and uses it to set the pitch of oscillator 1.

Try altering the WAVEFORM 3 command in line 30 to hear the shapes of the different waveforms (remember if you change it to WAVEFORM 3,3 to add line 35 PWM 3,2048). You may like to try to modify the ENV3 program to draw graphs of the different waveforms.

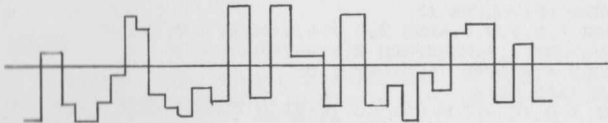
Shapes obtained



Pulse Wave



Wave is low for $\frac{P_{WM}}{40.96}$ % of the time.
White Noise



(random)

Music with SID - calculating PITCH values

Because SID has three voices, the pitch of which can be very accurately controlled (PITCH takes a value between 0-65535), and the sound output is so clean and 'musical', SID is ideally suited to creating computer music. Fortunately for us programmers, Commodore have designed SID so that the output frequency of its oscillators is directly proportional to the PITCH value.

- So:
- To go up an octave, double the PITCH value.
 - To get to the next note of an even-tempered scale, multiply by the 12th root of two. This is $2^{1/12}$ or 1.05946309.

Here is a program which lists all the note values:

```
10 Z=2^(1/12):A=274.93:J=0
20 REPEAT
30 READ Z$:PRINT Z$:"-";MID$(STR$(J),2),INT(A+.5)
40 IF Z$="B" THEN RESTORE:J=J+1
50 A=A*Z:UNTIL A>65535
60 DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B
```

Alternatively, here is a table of the top octave values - when you need lower ones, just divide by 2 the required number of times to get the same note but several octaves lower.

B-6	:	33216	:	F-7	:	46974	:
C-7	:	35191	:	F#-7	:	49768	:
C#-7	:	37284	:	G-7	:	52727	:
D-7	:	39501	:	G#-7	:	55862	:
D#-7	:	41849	:	A-7	:	59184	:
E-7	:	44338	:	A#-7	:	62703	:

Note that in line 30 of the above program, MID\$(STR\$(J),2) is

used instead of just J to remove the leading space.

Now, here's a demonstration music program which plays a short 3-part tune.

```
10 REM *** 3-PART 'HARPSICHORD' MELODY ***
20 DIM A(3,16),N(29)
30 REM-SET UP SID
40 SOUNDOFF:VOLUME 15
50 ADSR 1,0,9,0,0:ADSR 2,0,9,0,0:ADSR 3,0,9,0,0
60 WAVEFORM 1,2:WAVEFORM 2,2:WAVEFORM 3,2
70 FMODE 4:FILTER 7:CUTOFF 1500
80 REM-READ DATA
90 FOR Z=0 TO 3:FOR K=1 TO 16:READ A(Z,K):NEXT:NEXT
100 REM-WORK OUT PITCH VALUES
110 C=2↑(1/12):N(0)=4399:FOR K=1 TO 29:N(K)=N(K-1)*C:NEXT
120 DE=300
130 REM-PLAY NOTES
140 FOR K=1 TO 16
150 PITCH 1,N(A(0,K)):PITCH 2,N(A(1,K)):PITCH 3,N(A(2,K))
160 RELEASE 1:RELEASE 2:RELEASE 3:ATTACK 1:ATTACK 2:ATTACK 3
170 FOR J=1 TO DE*A(3,K):NEXT
180 NEXT K
190 REM-NOTE DATA
200 DATA 24,24,26,23,24,26,28,28,29,28,26,24,26,24,23,24
210 DATA 16,16,21,14,14,19,19,24,21,24,17,16,21,16,17,16
220 DATA 12,9,5,7,9,11,12,9,5,7,8,9,5,7,7,0
230 REM-DURATION DATA
240 DATA 2,2,2,3,1,2,2,2,2,3,1,2,2,2,2,6
```

Array A holds note and duration data - A(0,x) holds the notes for voice 1, A(1,x) the notes for voice 2, A(2,x) the notes for voice 3 and A(3,x) the duration of each chord. Array N holds the actual PITCH values.

Line 70 contains commands which you have not yet met to use the filter (these will be explained in the next chapter). These are only to change the tone quality of the music, and the program works perfectly well without them.

REMEMBER the steps for creating a sound:

- 1). SOUNDOFF
- 2). Set the VOLUME
- 3). Set the WAVEFORM, ADSR envelope and PITCH (in any order).
- 4). ATTACK - start the envelope.
- 5). To stop the sound, RELEASE the envelope.

Chapter 8

ADVANCED SOUND

Using the commands and functions described in the previous chapter, you will have been able to create very impressive music, and basic sound effects (e.g. sweeping the PITCH of a note, or of the noise). This chapter describes the facilities which put the 64 way above any other machine in its class.

Ring modulation

Ring modulation is a powerful sound facility, normally only found on expensive synthesisers, which produces non-harmonic overtone structures for use in mimicking bell or gong sounds, as well as creating various weird noises.

When ring modulation is selected, it replaces the triangle output of one of the oscillators with the 'ring modulated' combination of itself and another oscillator - technically, the waveforms are combined to give non-harmonic sum and difference tones.

Ring modulation is selected by the following command:

```
RINGMOD v,n  
(riN)
```

v is the voice of the oscillator concerned; n is non-zero to enable ring modulation or zero to disable it for that voice. An oscillator is always modulated by the oscillator BEFORE it numerically - i.e. oscillator 3 is ring modulated by oscillator 2, oscillator 2 by oscillator 1 and oscillator 1 by oscillator 3, the most common arrangement being the latter.

e.g. RINGMOD 1,1

Sets oscillator 1 to be ring modulated by oscillator 3. From now on, whenever the TRIANGLE output of oscillator 1 is selected, the ring modulated combination of oscillators 1 and 3 will be heard. Note that the only setting of voice 3 which has any effect is PITCH - there is no need to set a waveform or ADSR envelope for this voice.

Demonstration programs:

```
10 REM THE TARDIS!  
20 SOUNDOFF:VOLUME 15  
30 ADSR 1,0,0,15,0:ATTACK 1
```

```

40 WAVEFORM 1,1:REM SELECT TRIANGLE WAVEFORM
50 RINGMOD 1,1
60 PITCH 3,5000:REM BEING MODULATED BY OSC. 3
70 FOR K=0 TO 65535 STEP 250:PITCH 1,K:NEXT
80 FOR K=65535 TO 0 STEP -250:PITCH 1,K:NEXT
90 GOTO 70

```

Changing line 60 to PITCH 3,65535 gives a sound like tuning an old radio set.

The following program gives a detuned bell sound whenever you press 'X' - try playing around with the PITCH values in line 30.

```

10 SOUNDOFF:VOLUME 15
20 ADSR 1,0,9,0,0:WAVEFORM 1,1:RINGMOD 1,1
30 PITCH 1,17000:PITCH 3,25000
40 RELEASE 1:ATTACK 1
50 REPEAT GET A$:UNTIL A$="X"
60 GOTO 40

```

REMEMBER - The TRIANGLE output of the oscillator must be selected for the effect to be heard - other waveforms are unaffected.

Synchronisation

Another 'special effect', similar to RINGMOD. Synchronisation, as the name suggests, synchronises one oscillator to another - i.e. when one oscillator (the one being synchronised to) starts a cycle, the other is forced to start a cycle, no matter what part of the waveform it was in the process of outputting at the time. This creates complex harmonic structures and 'hard sync' effects.

Synchronisation is selected by the followig command:

```

SYNC v,n
(syN)

```

The format of this command is identical to RINGMOD - i.e. oscillator v is synchronised to the one numerically before it if n is non-zero.

e.g. SYNC 1,1

synchronises oscillator 1 to oscillator 3. Oscillator 3 must be set to some frequency other than zero, by PITCH, but preferably lower than the frequency of oscillator 1. Varying the frequency of oscillator 1 produces a wide range of complex harmonic structures from voice 1 at the frequency of oscillator 3. No other parameters of voice 3 have any effect on SYNC.

```

e.g.                10 SOUNDOFF:VOLUME 15
                    20 ADSR 1,0,0,15,0:ATTACK 1:WAVEFORM 1,2
                    30 PITCH 3,5000
                    40 SYNC 1,1
                    50 FOR K=0 TO 65535 STEP 100:PITCH 1,K:NEXT

```

```
60 FOR K=65535 TO 0 STEP -100:PITCH 1,K:NEXT
70 GOTO 50
```

This is very similar to the RINGMOD demonstration program for the Tardis.

Disabling the output from oscillator 3

Since oscillator 3 is used so commonly for modulation, Commodore have provided a bit in SID to disable its output, so that whatever waveform is being used for modulation is not heard on the audio channel. The command in BC BASIC which affects this bit is:

```
CH3,n
```

If n is zero, oscillator 3 is disabled; if n is non-zero oscillator 3 is re-enabled. SOUNDOFF enables the output of oscillator 3 automatically.

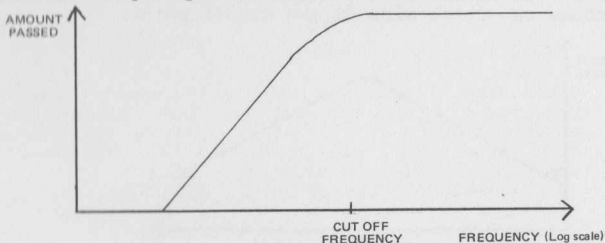
Using the filters in SID

Filtering is an extremely useful feature of SID which allows the timbre, or sound quality, of a sound to be altered - for example, to create tinny sounds, or to give a sound a muffled, bassy quality. Filtering is similar to using the tone controls on an amplifier, but much more flexible.

SID provides three filters, and two may be combined to create a fourth. We shall look at each in turn.

The high-pass filter

As its name suggests, this filter allows high frequencies to pass, but cuts the lower ones (like turning down the BASS control on an amplifier). If we look at a graph of the amount of signal passed against the frequency, it looks as follows:



As you can see, all signals above a certain point (the CUTOFF frequency) are left alone, but the signal is attenuated (cut) more and more as you go below the cutoff frequency. As bass frequencies are cut, the high-pass filter produces tinny, buzzy

sounds.

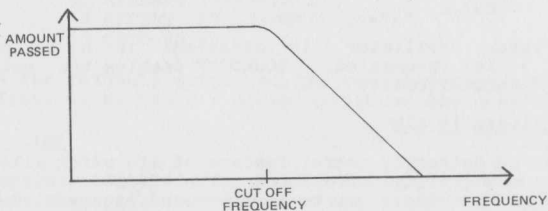
The cutoff frequency of all the SID filters is set by the CUTOFF command:

CUTOFF n
(cU)

n is between 0 and 2047, and corresponds to cutoff frequencies of 30 Hz to 12 kHz approximately.

The low-pass filter

This filter performs the opposite function to the high-pass filter - it lets low frequency signals pass, but attenuates those above the cutoff frequency.

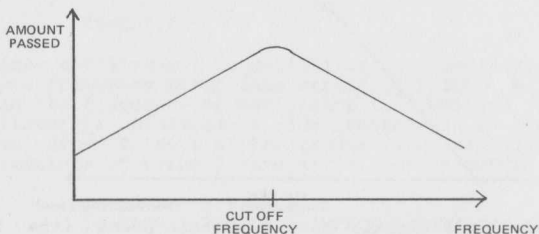


Since the treble frequencies are cut, the low-pass filter produces full-bodied, bassy sounds.

For the technically-minded, the low and high-pass filters have a cutoff rate of 12dB/octave.

The bandpass filter

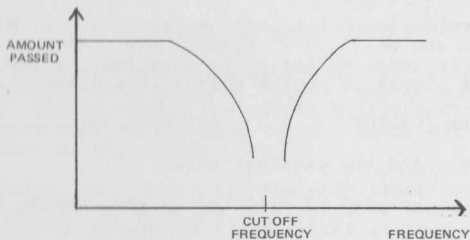
This filter allows sounds at the cutoff frequency to pass, but attenuates those on either side of the cutoff point.



The bandpass filter gives thin, open sounds. It has a cutoff rate of 6dB/octave on either side of the cutoff frequency.

The notch filter

The high and low pass filters can be combined to create a band reject or notch filter. This filter allows all frequencies to pass apart from a small band at the cutoff frequency.



Routing SID voices through the filter

Any or all of the SID voices may be selected to go through the filter, plus the signal from the external AUDIO IN input (more on that later).

Which voices are to be filtered is selected by the FILTER command:

FILTER n
(fI)

where n is between 0 and 15 and specifies which voices are to go through the filter; when n is taken in binary:

 3210

FILTER %xxxx (x=0 or 1)

Bit 0 is to filter voice 1, bit 1 is to filter voice 2, bit 2 is to filter voice 3 and bit 3 is to filter the external audio signal. If the bit is zero, filtering is disabled; if the bit is one, that particular signal is routed through the filter.

e.g. FILTER %1101 (=FILTER 13)

filters voices 1 and 3 and the external input.

The type of filter to be used is selected by the FMODE (filter mode) command:

FMODE n
(fM)

where n is between 0 and 7, and decides which filter is to be used, according to the following table:

: FMODE	: EFFECT	:
:-----:		
: 0	: Disables output from filter (all voices routed:	:
:	: through filter will be inaudible)	:
: 1	: Selects LOW-PASS filter	:
: 2	: Selects BAND-PASS filter	:
: 4	: Selects HIGH-PASS filter	:
: 5	: Selects NOTCH filter (5=4+1=high pass and low:	:
:	: pass combined)	:

Other values of n are just combinations of the three main filters.

For the filtering effect to be audible, one of the filters must be selected and at least one voice must be routed through it.

Resonance

Resonance is a peaking effect which emphasises frequency components at the cutoff frequency, causing a sharper sound. If you wish to use resonance, it is selected by the following command:

RESONANCE n

(res0)

n selects the amount of resonance. The resonance settings range linearly from no resonance (n=0) to maximum resonance (n=15).

Using the filters

An example of static filter settings is in line 70 of the music program at the end of the last chapter:

```
70 FMODE 4:FILTER 7:CUTOFF 1500
```

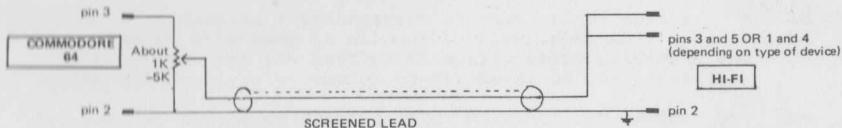
This sets the filter to HIGH-PASS (to get a tinny sound for a harpsichord simulation), routes all three voices through the filter (since FILTER 7 = FILTER %0111) and then sets the cutoff frequency to a fairly high value, to eliminate much of the bass component of the sound.

The filter is, perhaps, the most important element of the SID chip as it allows the generation of complex tone colours by subtractive synthesis - the filter is used to eliminate certain frequencies from an input signal rich in harmonics. The best results, however, are achieved by varying the CUTOFF frequency in real time, especially by tracking the ADSR envelope (the ENV3 function is invaluable here). There are enormous possibilities, from 'wah-wah' effects (try writing a keyboard organ program using voice 3, and continually setting the CUTOFF from ENV3) to wind and sea simulations.

Interfacing SID to the outside world

i) The AUDIO OUT line.

The output from the SID chip is available on the audio/video socket on the back of the 64, for improved sound quality and for recording purposes. Referring to page 142 of the Commodore 64 User Manual, connect pin 2, the GND line, to the EARTH of your amplifier and pin 3 to the signal input. The 64 has an output impedance of about 1K. Since the voltage output is fairly high, a potential divider may be required:



ii) The AUDIO IN line.

This is pin 5 on the audio/video connector. Signals received on this line will be sent to the audio output and the TV after being subjected to the VOLUME and FILTER settings (if filtering of this signal has been enabled by

FILTER). The signal should not exceed 3 volts p-p. Since the audio path from AUDIO IN to AUDIO OUT has unity gain when unfiltered, this input can be used to mix the outputs of several 64s by 'daisy-chaining' (linking each one into the next).

Chapter 2

ADVANCED SPRITES

This chapter deals with the advanced sprite facilities not covered in chapter 3.

Multi-colour sprites

Normal sprites have two colours - the SPRINK colour, and 'transparent' (the background is displayed). Multi-colour sprites allow three colours plus transparent (although two colours are common to all sprites) at the expense of halving the horizontal resolution.

When you use DEFSPR to define a multi-colour sprite, define it as twelve pairs of bits. The colours displayed are shown by the following table:

: BIT PAIR	:	COLOUR DISPLAYED	:
:-----	:	-----	:
: 00	:	Transparent	:
: 01	:	Sprite GCOL (graphics colour) 0	:
: 10	:	The SPRINK colour for that sprite	:
: 11	:	Sprite GCOL 1	:
:-----	:	-----	:

The sprite multi-colour graphics colours are set by:

```
SPRGCOL n,col
(sprgC)
```

where n is 0 or 1 (specifying which graphic colour is to be set).

So here is how the following sprite definition is displayed:

```
DEFSPR 13,0,%000001101010010011101100
.....AABBBBBBAA..CCBCC..
```

where '.' indicates a transparent section of the sprite, 'A' is the SPRGCOL 0 (common to all multi-colour sprites), 'B' is the SPRINK colour for the particular sprite being displayed and 'C' is the SPRGCOL 1, also common to all multi-colour sprites.

To put a sprite or sprites into multi-colour mode, use the SPRMODE command:

```
SPRMODE n[TO n2],z
or SPRMODE=n
(sprmO)
```

This command has the same formats as SPRON. The first format sets the mode of sprite n, or all sprites n to n2 inclusive - if z is zero, the specified sprite(s) are set to normal mode; if z is non-zero, the sprite(s) are set to multi-colour mode. The second format of this command sets the entire 'sprite multicolour select' register to n, which is 0 to 255. Each bit of n, when taken as a binary number, selects the mode of each sprite: a 0 is for normal mode and a 1 for multi-colour mode.

Functions are provided to read the multi-colour SPRCOLs and SPRMODEs:

SPRGCOL(n)	Returns sprite GCOL n (n is 0 or 1) as a number between 0 and 15.
SPRMODE(n)	Similar to the SPRON function, the first format returns 0 if sprite n is normal, -1 if it is multicolour; the second format returns the whole of the multi-colour select register as a number between 0 and 255.
or SPRMODE (sprm0)	

Sprite to background priorities

Normally, sprites pass in front of their background; however, they may be made to pass beneath it. This is set by the SPRPAPER command:

```

        SPRPAPER n[TO n2],z
or     SPRPAPER=n
        (sprpA)

```

The first format affects sprite n, or all sprites n to n2; if z is zero, the sprite(s) pass in front of text (normal setting) - if z is non-zero, then the sprite(s) are set to pass behind text. The second format sets the entire sprite-to-background priority register to n, which is 0-255 - a 0 bit is for normal setting, a 1 bit for sprites behind background.

There is no function to read the SPRPAPER setting - if you need to, use PEEK (\$D01B). There is a function SPRPAPER, but it is used to read sprite-to-background collisions; more on that shortly.

Sprites with hi-res

Sprites work totally normally with hi-res data, even to the point of passing under and over hi-res information and collisions being detected. The only real problem with displaying sprites with hi-res is the positioning in RAM of the sprite blocks.

When using hi-res screen 0, the RAM at \$8000-\$88FF, behind the BC BASIC ROM, is free, i.e. sprite blocks 0-47.

IMPORTANT NOTE - DEFSPR always takes a sprite block number from 0-255; where the sprite is positioned in memory will also depend on which VIC 16K block you are using. For example, when in MODE

3,0 (VIC block no. 2 at \$8000) DEFSPR 8 refers to $\$8000+8*64 = \8200 . Sprite blocks 253-255 are also free, but are cleared by CLG. AS A GENERAL RULE, ALWAYS SET UP THE HI-RES (MODE and CLG) before setting up sprites (i.e. defining them and setting sprite pointers).

Sprites are rather less easy to use when using screen 1 - only blocks 253 and 254 are free (and these are cleared by CLG). Block 47 is also free if REPEAT...UNTIL loops are not being used.

Interrupt-driven sprites

Now for one of the most interesting and exciting features of BC BASIC. Interrupt-driven sprites are a feature unique to the Commodore 64 running under BC BASIC.

What is involved is basically this: each sprite may be given a velocity, as well as a position. The sprite(s) concerned will then continue to move around the screen, without any further intervention from BASIC. They will even continue to move when your program has stopped. Any or all of the 8 sprites may be programmed to move, at speeds of up to 400 pixels per second in each of the x and y directions. This feature makes games writing easy!

Interrupt-driven sprites are controlled via 4 BC BASIC commands:

SPRGO n[TO n2],z
or SPRGO=n This command performs several functions. Firstly, it changes the speed of the interrupts from 60 times a second to 50 times a second (normally, the processor is 'interrupted' every 1/60th of a second, to scan the keyboard and update the clock). The purpose of this is to synchronise the interrupts to the TV, for really smooth movement. It also has the consequence, however, of slowing the clock down by 5/6ths, as well as other things such as the cursor flash rate. Secondly, this command changes the interrupt vector (the address to where the computer jumps every time it is interrupted) so that, as well as scanning the keyboard etc. the computer moves the sprites. Thirdly, this command changes the 'movement enable register' - i.e. which sprites are to be allowed to move. It has the same format as SPRON - e.g. SPRGO 3,1 enables movement of sprite 3
SPRGO 2 TO 5,0 disables movement of sprites 2-5 inclusive.
SPRGO=%10000100 enables movement of sprites 2 and 7 and disables movement of the others.

SPRSTOP This command neutralises the effect of SPRGO -
(sprsT) i.e. it resets the interrupt rate to 60Hz and

resets the interrupt vector to its normal position. It does not, however, affect the movement enable register. SPRSTOP is automatically called by SPRCLR, and by pressing RUN/STOP and RESTORE together. There is no function to read the movement enable register - if you need to, use PEEK(\$33A). SPRGO=PEEK(\$33A) can be used to restart movement after a SPRSTOP command.

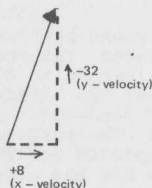
SPRV n[TO n2],xv,yv

This command is used to set the velocities of a sprite or sprites in the x and y directions. The sprite(s) affected are sprite n, or all sprites n to n2 if n2 is included. The x and y velocities of these sprites are set to xv and yv respectively - each may be between -127 and 127. Hence both velocity and direction may be specified.

Each sprite moves at $v \times 3.125$ pixels per second in the specified direction. For example:

SPRV 0,8,-32

makes sprite 0 move in the following direction if its movement has been enabled by SPRGO:



For sprites, a y position of 0 is at the top of the screen, so a negative y-velocity is UP, a positive one is down.

SPRTI n[TO n2],t

Another useful facility of BC BASIC interrupt-driven sprites is the ability to make certain sprites vanish and stop moving after a certain amount of time - useful for bullets etc. with a limited range.

Once again, either one sprite (n) or all sprites n to n2 are affected. t is the timeout-value, from 0-255, and is the number of 1/50ths of a second before the sprite(s) stop and vanish. If t is zero, timeout for the sprite(s) is disabled.

Once a sprite has stopped, it may be started again by: `SPRGO n,1:SPRON n,1` where n is the number of the sprite. The `SPRON(n)` function is useful for testing when a sprite has disappeared.

If you wish to read the current timeout value, use `PEEK($2B8+n)` where n is the number of the sprite.

Sprite velocities may be read by `PEEK($7E8+n)-128` for the x-velocity and `PEEK($7F0+n)-128` for the y velocity of sprite n.

Sprite collisions

This is the last sprite facility to be covered. After this will be a program demonstrating the commands introduced in this chapter.

A very useful aspect of the VIC chip is its collision detection circuitry, which will automatically detect collisions between sprites and their background, and sprites and other sprites.

Taking sprite-to-sprite collisions first:

Whenever non-transparent parts of two sprites are coincident (on top of each other) the appropriate bits are set in the sprite-to-sprite collision register - one bit for each sprite. These bits remain set (even if the sprites move apart again) until the register is read. When the register is read, ALL bits are set to zero, ready to detect more collisions. Of course, if the two sprites are still coincident, the bits in the collision register will be set again within 1/50th of a second, as the VIC chip gets round to displaying them on the screen again. Sprite collisions are even detected if the two sprites are off the screen.

The sprite-to-sprite collision register is read by `SPRSPR` function, which has two formats:

`SPRSPR` reads the entire collision register in one go, returning a value between 0 and 255. This sets the register to 0, but if the sprites are still coincident the register will be set again within a few lines of BASIC (1/50th of a second's worth of program).

or `SPRSPR(n)` reads the sprite to sprite collision register, returning 0 if bit n is not set, -1 if it is set (i.e. -1 if sprite n has collided with another sprite). THIS FUNCTION ALSO CLEARS THE COLLISION REGISTER, like `SPRSPR` on its own, so if you need to see which sprite sprite n has collided with, better to use, say, `A=SPRSPR` and then test A separately.

NOTE: after the sprite initialisation and just before the main loop in your program, put the line A=SPRSPR (A is a dummy variable). This is to clear the collision register of any collisions which may have happened during the setting up process, which would cause spurious collisions to be detected. Remember that the collision bits stay set, even if the sprites aren't in contact any more, until the collision register is read.

SPRITE-TO-BACKGROUND COLLISIONS are almost identical to sprite-to-sprite collisions. They are read using the SPRPAPER function, which has the same formats as SPRSPR, and the same rules that the register is cleared when it is read apply. The only difference is that, in multi-colour mode (MODE 1 or 4 - you have not met MODE 1 yet) collisions with colour number 1 (bit pair '01') are not detected. So, in MODE 4, plotting in colour 1 will not give collisions. This is to allow a background display without interfering with true sprite collisions. Just as with sprite-to-sprite collisions, the collision bit remains set even if the sprite concerned moves away from the screen data it collided with - the bit is only cleared when the collision register is read.

Here is a demonstration program using most of the features of BC BASIC introduced in this chapter. It is a game (use joystick in port 2) which would normally have taken pages of machine-code. A point to note: when you hit one of the asteroids, sometimes a different one vanishes. This is not a bug in the program as such, but it occurs when two asteroids are colliding and the bullet and a different asteroid are touching too - then, four bits are set in the collision register, and the program does not know which one to explode. To avoid this in your own programs, either don't make other things collide, or if they do, make them explode themselves.

```

1    REM *** INITIALISATION
5    MODE3,1:CLG
10   SPRCLR:FORK=0TO20:READA$:DEFSPR47,K,VAL("$"+A$):NEXT
20   FORK=0TO20:DEFSPR253,K,0:NEXT:DEFSPR253,13,$00F000
30   FORK=0TO20:DEFSPR254,K,$FFFFFF:NEXT:DEFSPR254,0,$3FFFFC:DE
    FSPR254,20,$3FFFFC
40   SC=0:HPRINTATO,0;#1;"SCORE: 0"
42   REM *** STARRY BACKGROUND
43   FORK=1TO35:X=RND(1)*319:Y=RND(1)*190:PLOTX,Y:PLOTX+1,Y:PLOT
    X,Y+1
44   PLOTX+1,Y+1:NEXT
47   REM *** MORE SPRITE SETUP
50   SPRON=$11111101:SPRPOINT0,47:SPRPOINT1,253:SPRPOINT2TO7,254
60   XV=0:YV=0:SPRPOS0,170,135:SPRV0TO1,0,0
70   FORK=2TO7:SPRPOSK,RND(1)*512,RND(1)*256:SPRVK,RND(1)*64-32,
    RND(1)*64-32
80   SPRINKK,K+1:NEXT
85   FORK=1TO1000:NEXT
90   SPRGO=255:CF=LOG(2):SP=3:A=SPRSPR
95   REM *** MAIN LOOP
97   REM *** ACCELERATE SHIP

```

```

100  XV=XV+JOYX(2)*SP:IFXV<-127THENXV=-127
110  IFXV>127THENXV=127
120  YV=YV+JOYY(2)*SP:IFYV<-127THENYV=-127
130  IFYV>127THENYV=127
140  SPRV0,XV,YV
150  IF JOY(2,4)=0ORSPRON(1)THEN170
155  REM *** FIRE!
160  SPRPOS1,SPRX(0)+20,SPRY(0):SPRON1,1:SPRV1,100,0:SPRT11,30:
    SPRGO1,1
165  REM *** CHECK FOR COLLISIONS
170  A=SPRSPR:IF(AAND1)THEN2000
180  IF(AAND2)THEN1000
190  GOTO100
995  REM *** BULLET HIT
1000 S=LOG(A-2)/CF
1010 SPRPOSS,RND(1)*512,RND(1)*256:SPRVS,RND(1)*64-32,RND(1)*64-
    32
1020 SC=SC+100:HPRINTAT0,12;#1;SC
1030 SPRON1,0:A=SPRSPR:GOTO100
1995 REM *** SHIP HIT
2000 SPRSTOP:X=SPRX(0):Y=SPRY(0):SPRV0,0,0
2005 SPRPOINT 4TO7,253:SPRON=1:FORK=0TO7:SPRPOS,X,Y:NEXT:
    SPRINK4TO7,1
2010 V=40:SPRV4,-V,-V:SPRV5,V,-V
2020 SPRV6,-V,V:SPRV7,V,V
2030 SPRT10TO7,25:SPRON=%11110001:SPRGO=%11110001
2040 FORJ=1TO3000:NEXT:RUN
8995 REM *** SPRITE DATA
9000 DATA0,0,0,0,0,0,0,78000,1FE000,F63000,F63800,7FFC00,7FFC00
    FFFFFF
9010 DATAIFF800,0,0,0,0,0,0

```

Chapter 10

MORE STRUCTURED PROGRAMMING

Procedures were fully dealt with in chapter 5; however, BC BASIC provides other programming structures to make programs easier to write and understand.

IF...THEN...ELSE

The format of a normal IF statement is as follows:

IF condition THEN statements1

If the condition is TRUE (non-zero), 'statements1' are executed and then execution continues on the next line. If the condition is FALSE, 'statements1' are ignored and program execution proceeds directly onto the next program line.

BC BASIC allows a modified format of IF:

IF condition THEN statements1:ELSE statements2
(eL)

If the condition is TRUE, then 'statements1' are executed and then execution moves to the next line; if the condition is false, 'statements2' are executed and then execution moves to the next line.

ELSE on its own acts like a REM statement - it ignores everything following it. Like THEN, ELSE may be followed just by a line number without a GOTO:

e.g. IF A=1 THEN 50:ELSE 200

REPEAT...UNTIL

This is a form of loop, which continues to loop round UNTIL a certain condition is met. The loop is started by the command REPEAT, which does not have to be followed by a colon, and the end of the loop is marked by UNTIL followed by the condition.

e.g. 10 REPEAT:GET A\$:UNTIL A\$="X"
 (reP) (uN)
 continues to loop until an X is pressed.

REPEAT.....UNTIL 0 is a continuous loop, since 0 is never true.

REPEAT...UNTIL loops may be nested (placed one inside the other) up to 16 deep, after which an ?OUT OF MEMORY error will occur. UNTIL without REPEAT gives an ?UNDEF'D STATEMENT error. The REPEAT...UNTIL stack (the memory where the computer stores the positions of REPEAT instructions) is cleared by RUN and CLR - so don't use CLR inside a REPEAT..UNTIL loop!

RUN and CLR also clear the procedure stack, so don't use CLR within procedures either. Remember also that HIMEM calls CLR, so this too will clear REPEAT...UNTIL loops and all procedures.

REPEAT...UNTIL loops are useful for generating continuous loops in immediate mode, but be careful when you set up an infinite REPEAT...UNTIL loop in immediate mode, break out of it, and then start it again - if you keep doing this you will eventually fill the REPEAT...UNTIL stack and get an ?OUT OF MEMORY error because the computer thinks you have 'nested' your loops too deep. Typing CLR will solve the problem.

A closer look at procedures - recursion

Because you can make BC BASIC procedures completely self-contained, by using parameter lists, local variables and not using FOR...NEXT loops in them, they are ideally suited to writing recursive routines. Recursion is a powerful programming technique far too complex to discuss here, but will be familiar to expert programmers. It enables problems like the 'Towers of Hanoi' to be solved using a program just a few lines long. Here is a program which will print all the combinations of a word up to 17 letters long:

```
10 INPUT"WORD";W$
20 PROC SCRAMBLE("",W$)
30 END
100 DEFPROCSCRAMBLE(R$,W$):LOCAL K
110 IF LEN(W$)<2 THEN PRINT R$;W$;ENDPROC
120 K=1
130 REPEATPROCSCRAMBLE(R$+MID$(W$,K,1),LEFT$(W$,K
-1)+MID$(W$,K+1))
140 K=K+1:UNTIL K>LEN(W$)
150 ENDPROC
```

e.g. running this program and typing CAT would give:

```
CAT
CTA
ACT
ATC
TCA
TAC
```

It is limited to words of 17 letters because of the use of REPEAT...UNTIL loops - if an IF statement had been used instead, words of up to 23 letters could be used (it is then limited by the size of the procedure stack).

Chapter 11

MORE I/O (Input/Output)

This chapter deals with the advanced input/output facilities of BC BASIC.

Function keys

BC BASIC allows the user to assign commands to each of the 8 function keys, F1 to F8. These keys are to the right of the keyboard: F1, F3, F5 and F7 are unshifted, and F2, F4, F6 and F8 are obtained by pressing SHIFT with the appropriate key.

To define a function key, use the KEY command:

```
KEY n,"definition"  
(kE)
```

n is the number of the key (1-8) and "definition" is the string which will be assigned to that key. The length of a definition is limited to the length of the keyboard buffer (normally 10 characters) - any more than this is ignored. The size of the keyboard buffer is held in \$289, and may be increased to 15 characters if you are not using the RS232 port, by POKE \$289,15. Whatever happens, DO NOT increase it beyond 15 characters!

To list the definitions of all the keys, use:

```
KEYLIST  
(kEI)
```

If you wish, you may then move the cursor over one of the definitions to change it.

A function is provided to read a certain function key:

```
KEY$(n)  
(kE$)
```

This function returns, as a string, the definition of function key n. It is used by BASIC programs to see what a key is defined as. Since the function keys are not cleared by CLR, RUN or even LOAD, a novel use of this facility would be to pass data, as 10-character strings, between programs, or storing hi-scores etc.

The function keys are active in immediate mode and during an INPUT, but not when GET is being used (GET will return the ASCII code of the function key, between 133 and 140, rather than its definition).

As an aside, you may be interested to know that all the keys (including the function keys) may be made to auto-repeat by POKE 650,128. To return to normal, POKE 650,0.

Here are some examples of useful function-key definitions:

```
KEY 1,"LIST"+CHR$(13)   (CHR$(13) = 'RETURN')
KEY 2,"LOAD"+CHR$(34)   (CHR$(34) = double-quote)
KEY 4,CHR$(34)+",8"+CHR$(13)
```

To load a program from disk, just type **[F2]** filename **[F4]**.

Games paddles

A pair of games paddles, or one analogue joystick, may be plugged into each games port. They have the advantage over conventional joysticks of allowing a range of positions, rather than just having four switches, each of which may be either 'off' or 'on'. The games paddles, and their fire buttons, are read by the PADDLE function.

```
PADDLE(n[,z])
(paD)
```

n is between 1 and 4, and specifies which paddle is to be read:

```
-----
: n      Paddle read      :
:-----:
: 1      Paddle X, port 1  :
: 2      Paddle Y, port 1  :
: 3      Paddle X, port 2  :
: 4      Paddle Y, port 2  :
:-----
```

z, if present, must be 0 or 1, and is assumed to be 0 if left out. 0 tells BC BASIC to read the value of the paddle, returning a number 0-255, which is proportional to the resistance of the paddle at that moment. If z is 1, the FIRE button status is read - 0 if the switch is not pressed, -1 if it is.

If you wish to make your own paddles, you should connect a 470K linear potentiometer between the +5V pin, pin 7, of the game I/O port required and either pin 9 (POT X) or pin 5 (POT Y). Pins 3 and 4 are the two fire buttons, and should be connected via normally-open switches to GND (pin 8) if they are required. See p.141 of the Commodore 64 User Manual for pinouts. A suitable connector is a 9-way D-range socket, e.g. Maplin RK61R.

```
e.g.
10 MODE 3,0:CLG
20 CLG:FOR X=0 TO 319
30 DRAW TO X,PADDLE(1)*.75+5
40 IF PADDLE(1,1) THEN 20
50 NEXT X:GOTO 20
```

This plots the position of PADDLE 1 as a graph, clearing the screen whenever FIRE is pressed.

BC BASIC has a function, KEY (), to scan an individual key to see if it is pressed, regardless of any other keys being pressed. This feature is useful when writing two-player games without joysticks, or to allow diagonal movement by pressing the 'up' and 'left' keys together, for example.

KEY (n)
(kE)

This function returns either 0 or -1; 0 if the key is not pressed, -1 if it is. n specifies which key is to be tested, and is in the range 0-63; it is the code of the key read by PEEK(197).

```
e.g. REPEAT PRINTPEEK(197):UNTIL0
```

If you type this in, a stream of numbers will appear on the screen. Pressing 'A' will make '10' appear; pressing 'S' will make '13' appear. The codes of the other keys may be found in this way. Now:

A=KEY(10) will set A to 0 if the key 'A' is not being pressed, -1 if it is.

A=KEY(13) will set A to 0 if the key 'S' is not being pressed, -1 if it is.

A=A+KEY(10)-KEY(13)
will subtract 1 from A if 'A' is
pressed, and add 1 to A if 'S' is
pressed (this has obvious
implications for games).

The code for 'W' is 9, and the code for 'Z' is 12.

```
Now:      10 PRINT"[cls]":X=20:Y=12
          20 PRINT AT Y,X;"*"
          30 FOR K=1 TO 25:NEXT
          40 PRINT AT Y,X;" "
          50 X=X+KEY(10)-KEY(13):Y=Y+KEY(9)-KEY(12)
          60 GOTO 20
```

This moves a star around the screen, using the keys W,A,S and Z. Pressing two keys together (e.g. W and A) will move the star diagonally.

Here is a table of KEY values:

: 0 : DEL	: 16 : 5	: 32 : 9	: 48 : £	:
: 1 : RETURN	: 17 : R	: 33 : I	: 49 : *	:
: 2 : CRSR →	: 18 : D	: 34 : J	: 50 : ;	:
: 3 : F7	: 19 : 6	: 35 : 0	: 51 : HOME	:
: 4 : F1	: 20 : C	: 36 : M	: 52 :	:
: 5 : F3	: 21 : F	: 37 : K	: 53 : =	:
: 6 : F5	: 22 : T	: 38 : O	: 54 : ↑	:
: 7 : CRSR ↓	: 23 : X	: 39 : N	: 55 : /	:
: 8 : 3	: 24 : 7	: 40 : +	: 56 : 1	:
: 9 : W	: 25 : Y	: 41 : P	: 57 : ←	:
: 10 : A	: 26 : G	: 42 : L	: 58 :	:
: 11 : 4	: 27 : 8	: 43 : -	: 59 : 2	:
: 12 : Z	: 28 : B	: 44 : .	: 60 : SPACE	:
: 13 : S	: 29 : H	: 45 :	: 61 :	:
: 14 : E	: 30 : U	: 46 : @	: 62 : Q	:
: 15 :	: 31 : V	: 47 : ,	: 63 : STOP	:

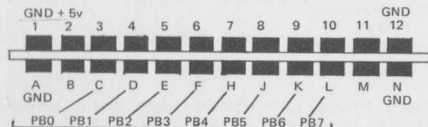
KEY codes 15,52,58 and 61 have no key associated with them. ASCII codes of KEY codes may be read by PEEK(\$EB81+n) where n is the KEY code (0-63).

NOTE: Because of the way the 64's keyboard circuitry works, certain combinations of 3 keys actually make the 64 think it can see a fourth one as well. If this causes problems, use a different set of keys in your program.

The user-port

The USER PORT is a useful input/output facility of the Commodore 64, which is well worth experimenting with if you have a basic grounding in electronics. It can be used in all sorts of control and sensing applications, e.g. robot control, counting, burglar alarms etc. The user port is the large edge-connector on the back of the 64, next to the cassette port. A suitable connector for it is available from your Commodore dealer.

This is the pinout of the user port:



Lines PBO to PB7 are the ones we are interested in. Each may be either an input or an output: this is set by the PORT command.

PORT~~n~~
(poR~~n~~)

n, in binary, represents the direction of each line on the user port. It is between 0 and 255 (%00000000 and %11111111). A 0 bit indicates that the line is to be an input, a 1 bit makes the line an output.

765434210

e.g.

PORT~~%~~00001101

sets bits 0,2 and 3 of the user port as outputs,
the rest as inputs.

Pressing RUN/STOP and RESTORE automatically does a PORT 0, i.e. sets all the lines to inputs.

The lines which have been defined as outputs may now be set to be either LOW (OV) or HIGH (+5V). This is set by the PORT command, which has the same formats as SPRON:

PORT n[TO n2],z
or PORT=n
(poR)

The first format affects bit n, or all the bits n to n2. If z is 0, the bits are set to a LOW (OV) value; if z is non-zero, the bits are set to HIGH (+5V).

e.g.

PORT 3 TO 5,1

will set PB3 to PB5 to +5V if
they have been defined as
outputs.

The second format sets all the lines of the user-port which have been defined as outputs in one go. n is 0-255 - in binary, a 0 bit is for a LOW output, a 1 bit for a HIGH output.

The user port may be read by the PORT function, which has the same formats as the SPRON function:

PORT
or PORT(n)
(poR)

The first format reads the whole user port in one go, giving a value 0-255, where each bit is a 0 for a LOW or 1 for a HIGH.

e.g.

if PRINT BIN\$(PORT,8)

prints 00011011, then PB2
,5,6 and 7 are at OV and
PBO,1,3 and 4 are at +5V.

The second format reads one bit, bit *n*, where *n* is 0-7. It returns 0 if the bit is low, and -1 (logical 'TRUE') if the bit is high.

If PORT is used to read a line or lines which have been defined as outputs, the value read will be the value which the line has been set to by the PORT command.

The DDR (data direction register) of the user port may also be read, using the PORT<function>, which has the same formats as the PORT function. It is used to see which lines have been defined as inputs, and which as outputs.

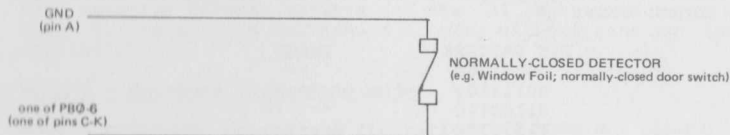
Here is a program using the PORT commands and functions:

```

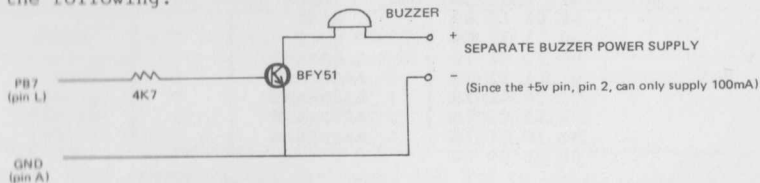
10 REM BURGLAR ALARM
20 REM SENSE INPUTS ARE BITS 0-6
30 REM ALARM OUTPUT IS BIT 7
40 PORT←%10000000
50 PORT=0
60 FOR K=0 TO 6
70 IF PORT(K) THEN PRINT"INTRUDER,AREA";K:PORT7,
  1:GOTO60
80 NEXT K:PORT 7,0:GOTO 60

```

Lines on the user-port which are defined as inputs but not connected to anything float up to +5V (HIGH) and return '1'. So connect the following to each PB line:



Each line is normally connected to GND (i.e. 0V) - but when the contact is broken (e.g. by smashing a window) the input line floats to +5V, and the program recognises this by setting PB7 to +5V (in line 70) and printing which contact has been broken. Obviously, a circuit is needed to sound the alarm from PB7, like the following:



There is an extra advantage to this circuit - if RUN/STOP and RESTORE are pressed, PB7 is then defined as an input, so it floats up to +5V and sounds the buzzer!

Chapter 12

ADVANCED CHARACTER GRAPHICS

This chapter covers MODE 1 and MODE 2, the two graphics modes not yet mentioned.

MODE 1 - Multi-colour character mode

This mode is similar to MODE 0 (standard character mode) - however, each character may contain up to 4 colours (including the background), with a loss of resolution, and only the first 8 colours (Black to Yellow) may be used as foreground colours.

When MODE 1 is selected, each individual character square may be set to be normal or multi-colour - this is selected by bit 3 of colour memory. In other words, selecting a colour of 0 to 7 makes that character normal, just as if it were being displayed in mode 0. However, if its colour is 8 to 15, it is displayed in multi-colour mode. This is to allow mixing of hi-res characters with multi-colour characters on the same screen.

Once multi-colour mode has been selected for a character, each PAIR of bits determines which colour each pair of dots on the screen will be displayed as. Take the example of the letter A, in normal mode:

BIT PATTERN	IMAGE
00011000	**
00111100	****
01100110	** **
01111110	*****
01100110	** **
01100110	** **
01100110	** **
00000000	

In MODE 1, when bit 3 of its colour 'nybble' (1 nybble = 1/2 a byte) is set, it is displayed as follows:

BIT PATTERN	IMAGE
00 01 10 00	AABB
00 11 11 00	CCCC
01 10 01 10	AABBAABB
01 11 11 10	AACCCBBB
01 10 01 10	AABBAABB
01 10 01 10	AABBAABB
01 10 01 10	AABBAABB
00 00 00 00	

As you can see, each pair of bits determines the colour of a pair of dots (01=AA, 10=BB, 11=CC). The colour displayed for each bit comes from the following table:

BIT PAIR	COLOUR DISPLAYED	SET BY	
00	Background 0 (PAPER)	PAPER col	
01	Background 1	SETCOL 1,col *	
10	Background 2	SETCOL 2,col	
11	Lower 3 bits of colour RAM:	INK col+8	

* NOTE: SETCOL 1,col also sets the INK to col, so you will need to reset the INK after using this command. SETCOL 0,col may be used instead of PAPER col.

Note that you are not putting the actual colour codes in the bits of your character, just references to the registers where these are stored. This conserves memory (2 bits to select 16 colours) and makes some neat tricks possible - changing the appropriate register will change every pair of dots on the screen drawn in that colour.

MODE 1 allows several colours to be present in one character, but places restrictions on the numbers of colours used and on resolution - each character is only 8 by 4 dots, horizontal dots being twice as wide as before. MODE 1 is best used with user-defined characters.

SPRITE COLLISIONS IN MODE 1 - sprite-to-background collisions are not detected between sprites and the '01' multicolour bit pair. This is to allow a background display without causing spurious collisions.

MODE 2 - Extended background mode

MODE 2 overcomes one of the limitations of MODE 0 - namely, that there is only one background colour for the whole screen - but at the expense of the numbers of characters which may be displayed.

In Extended Background Mode, the lower 6 bits of the character code determine which character will be displayed - therefore, only characters 0-63 are available in this mode. The top two bits determine from what register the background colour comes:

CHARACTER CODE			BACKGROUND COLOUR REGISTER	
RANGE	BIT 7	BIT 6	NUMBER	SET BY
0-63	0	0	0	PAPER col or SETCOL0,col
64-127	0	1	1	SETCOL 1,col *
128-191	1	0	2	SETCOL 2,col
192-255	1	1	3	SETCOL 3,col

* NOTE: SETCOL 1,col also sets the INK to col.

For example: in MODE 2, POKEing character code 1 to the screen will display the character 'A' in the PAPER colour, as expected.

However, poking 65 will still display an 'A', not a 'spade' (shift-A) symbol, but it will be in the BACKGROUND 1 colour. Similarly 129 and 193 - they will both display an 'A', but in BACKGROUND colours 2 and 3 respectively.

This mode is useful for highlighting text etc. when graphics are not required, or when user-defined characters are being used and not more than 64 of them are needed.

WHENEVER THE COMMANDS MODE 0, MODE 1 AND MODE 2 ARE EXECUTED, the CHARSET is reset (as if the command CHARSET 2,1 had been executed) - so when using UDGs, it may be necessary to follow each MODE command by a CHARSET command.

Notes on flicker-free movement using PRINT AT

The problem encountered with movement in chapter 1 was that it was slowed down by a delay loop; if the delay was removed, however, the character flickered badly. The solution to this problem is to move the cursor to the position of the old character, do a SCRWAIT (to synchronise to the TV blanking period) then print a space over the old character and print the new character somewhere else. This may be performed in one of two basic ways:

```
i)          10 PRINT "[cls]":X=20:Y=12
             20 SCRWAIT:PRINT"[crsr-left][space]";AT Y,X,"*";
             30 X=X+JOYX(2):Y=Y+JOYY(2):GOTO 20
```

This program relies on the fact that, after the PRINT AT operation in line 20, the cursor is left on the space to the right of the star - so moving the cursor left and printing a space will erase it. [crsr-left] in line 20 actually means shift and [crsr-left] - otherwise it would be a crsr-right. This method will obviously only work if you are moving one character around - the following, however, can be expanded to move several characters.

```
ii)         10 PRINT"[cls]":X=20:Y=12
             20 AT OY,OX:SCRWAIT:PRINT" ";AT Y,X,"*";
             30 OX=X:OY=Y:X=X+JOYX(2):Y=Y+JOYY(2):GOTO 20
```

This program keeps the old position of the star in variables OX and OY.

Both these programs use a joystick in port 2; if you do not have one, replace:

```
X=X+JOYX(2):Y=Y+JOYY(2)
```

with `X=X+KEY(10)-KEY(13):Y=Y+KEY(9)-KEY(12)`

Controls are then W,A,S and Z.

Chapter 13

MACHINE-CODE PROGRAMMING AIDS

BC BASIC provides an impressive set of commands and functions to assist the machine-code programmer, some of which, like \$ and HIMEM, you have met already.

2 and 4-byte equivalents of PEEK and POKE

BC BASIC allows you to PEEK or POKE 2 or 4 bytes at once (bytes in reverse order), for setting and reading pointers or passing information between BASIC and machine-code programs.

'n=z (single-quote, shift+7). This command puts the two-byte unsigned number z (0-65535) into the specified address and the byte following it, in low-high order. n may be a number or a variable, like 43 or K - if you need to use an expression, like K+2, then it must be put in brackets.

e.g. '43=\$0801 puts 1 in location 43 and 8 in location 44.

'n
or '(exp) This function, which is complementary to the ' command, returns the contents of the specified memory location, plus 256 times the contents of the following memory location (i.e. a low-high order read) as a number between 0 and 65535. It is used to read the contents of 2-byte pointers. As with the ' command, this function must be followed by a number or variable, or an expression in brackets.

e.g. PRINT '43 prints the bottom-of-memory pointer.

!n=z
or !(exp)=z This command, similar to the ' command, takes the 4-byte number z which may be between \$-FFFFFFFF and \$+FFFFFFFF, and stores the bytes in reverse order starting at location n (number or variable) or (exp). It is useful for the communication of large numbers to machine-code programs, and can be used to fill large areas of memory quickly - about 4 times faster than POKE.

e.g. !43=\$12345678 stores \$78 at 43, \$56 at 44, \$34 at 45 and \$12 at 46.

!n
or !(exp) This function reads 4 bytes of memory, stored in low to high byte order from location n or (exp), and builds them into one number. The bytes are read in the same way that they were stored using the ! command. N.B. Numbers are taken in two's complement form, so \$FFFFFFFF is read as -1.

Note that in the above 4 definitions, `n`, as well as being a number or variable, may be a function - so `SQR(9)` is the same as `'3` or `'(SQR(9))`, and `!''43` is the same as `!(''43))`. In this way, multiple indirection is possible - `PRINT ''43` prints the contents of the two-byte location pointed to by 43 and 44.

Hex and binary conversion

Some of this you will have met before - however, here are complete definitions of these functions.

`%[+/-]` binary digits

The `%` function takes the 0 and 1 digits following it, up to the first digit which is neither a 0 nor a 1. Note that the optional sign comes between the `%` and the binary digits - normally you may put the sign before the `%`, but this will not work in VAL. The string of binary digits may be a maximum of 32 characters long, excluding leading zeros, and decimals and exponents are not possible. As mentioned, `%` may be used in VAL, so string variables may be converted to decimal. The main use of `%` is in planning graphics (user-defined characters and sprites) where a string of 0s and 1s correspond to spaces and dots on the screen.

e.g. `PRINT %110101` prints 53
`A=VAL("%"+A$)` converts A\$ from binary to decimal in A.

`$[+/-]` hex digits

This function takes hex digits (0-9, A-F) following the `$` sign and converts them to decimal. Similar to `%` - may be used in VAL to convert hex strings to decimal, and no decimal points or exponents are allowed. Maximum of 8 digits, excluding leading zeros.

e.g. `PRINT $12345` prints 74565
`A=VAL("$"+A$)` converts A\$ from hex to decimal in A.

NOTE: Hex numbers, because less calculation is involved in converting them into a form that the computer can store, are much faster than decimal numbers - almost as fast as variables, in fact - and should be used in preference to decimal numbers where speed is important.

e.g. `POKE K,$FF` is much faster than `POKE K,255`.

@n or @n(exp[,d[,z]])

This function converts n (number/variable/function) or exp, which are in the range \$FFFFFFF to \$FFFFFFF into an ASCII string of hex characters, which starts with a space or a '-' sign, depending on the sign of the number. d, if present, must be in the range 0-8, and specifies the minimum number of digits in the answer, which will be padded with zeros if necessary. z, which must be in the range 0-255 specifies what character will replace the leading zeros.

e.g. PRINT @1234 prints 4D2
 A\$=@(617*2,4) sets A\$ to " 04D2"
 B\$=@(44000,7,42) sets B\$ to " ***ABE0"

BIN\$(exp[,d[,x,y]])
(b1)

This function converts exp, which must be in the range \$-FFFFFFF to \$+FFFFFFF, to a binary string, which starts with either a space or a '-', depending on the sign of exp (exp may be a number, variable, function or expression). d (0-32), if present, specifies the minimum number of binary digits in the answer, which will be padded with zeros if necessary. x and y, which are 0-255, are the ASCII codes of characters which will be substituted for 0 and 1 digits respectively. This last facility is useful when displaying sprites and UDGs as a series of characters on the screen - spaces and shift-Qs look much better than 0s and 1s.

e.g. PRINT BIN\$(234) prints 11101010
 A\$=BIN\$(53,8) sets A\$ to " 00110101"
 B\$=BIN\$(53,8,46,42) sets B\$ to " ..*.*.*"

Note that the abbreviation, bI, includes the \$.

Setting and reading the top-of-memory pointer

HIMEN=n
(h1)

n, 0-65535, is the address of the first memory location which may not be used by BASIC or the Kernal. As well as setting 55/56, the BASIC top-of-memory pointer, HIMEM calls the Kernal MEMTOP routine to move the RS232 buffer etc.

HIMEM PERFORMS A CLR and so should be used as the first statement in a program, or when the values of variables are not needed and no REPEAT...UNTIL loops or procedures are in operation. It should not be used while the RS232 port is in operation - only before the channel is opened, or after it has been closed.

HIMEM This function reads the top-of-memory pointer,
(hI) returning a value 0-65535, which is the first
memory location inaccessible by BASIC.

Saving a block of memory

```
MSAVE s,e[,][ "filename"[,dev[,sa]]]  
(mS)
```

This command saves a block of memory from s to e inclusive (both are addresses 0-65535). "filename", if specified, is a string expression for the filename; dev, if specified, is the device number (1 is assumed); sa, if specified, is the secondary address (0 is assumed if not present). The format from "filename" onwards is the same as an ordinary SAVE.

MSAVE is very useful when developing machine-code programs. Memory saved using MSAVE should be loaded by:

```
LOAD "filename",1,1                    from tape
```

```
or                    LOAD "filename",8,1                    from disk
```

If this is done in immediate mode, the top of program pointer will be altered, so you will need to type NEW. If it is done in program mode, the program will restart from the beginning, although with variables intact.

e.g. MSAVE \$7E00,\$7FFF,"M/C PROG1" saves memory from \$7E00 to \$7FFF inclusive to tape.

@ - user-defined command

Standard 64 BASIC allows you to define your own function,USR - BC BASIC also allows you to define your own command, which is called by the command @.

Whenever an '@' command is executed (NB. not the @ function!) a JSR is made to the address held in \$2FE and \$2FF. Typing:

```
PRINT @'$2FE
```

will show that this is normally \$AF08, the address of the BASIC ?SYNTAX ERROR message routine. However, this may be changed by:

```
'$2FE=$xxxx
```

Then, whenever @ is used as a command (i.e. at the beginning of a line, or after a ':'), a jump will be made to your own command processing routine at \$xxxx. Terminate your command with an RTS instruction, just as it it had been called by SYS \$xxxx.

By testing for the characters which come after the '@', you may add a whole series of commands to BC BASIC (just like there are many sprite commands which begin with SPR).

Chapter 14

MISCELLANEOUS USEFUL FACILITIES

Here are some useful facilities, found in the BASICs of some other machines, which have been implemented in BC BASIC.

RESTORE [exp] (res) The RESTORE command performs the same function as in standard 64 BASIC, i.e. to reset the DATA pointer, used by READ, to the beginning of the program. However, it is now much more powerful in being able to take a numeric expression to determine just where the pointer will be set to, i.e. which line. The specified line must exist, however, or an ?UNDEF'D STATEMENT error will result.

RESTORE exp is very useful in allowing the programmer to access large amounts of information in DATA statements, without having to read it into arrays first, which is slow and wasteful of memory.

e.g.

```
10 INPUT A
20 RESTORE A*10+90
30 READ Z$
40 PRINT Z$
50 GOTO 10
100 DATA FRED BLOGGS
110 DATA JOHN SMITH
120 DATA B C COMPUTERS
etc.
```

Typing a number with this program will read and print that piece of data.

**GOTO(exp)
GOSUB(exp)**

The two commands GOTO and GOSUB have now been improved by allowing the user to put a numeric expression in brackets after the command to specify which line to jump to. Like a normal GOTO or GOSUB, the line must exist, or an ?UNDEF'D STATEMENT error will result.

e.g.

```
230 PRINT "WHAT ACTION SHALL WE TAKE"
240 INPUT "1,2 OR 3";Z
250 IF Z<1 OR Z>3 OR Z<>INT(Z) THEN 240
260 GOTO(Z*100+200)
300 REM ACTION 1
.
.
400 REM ACTION 2
.
.
500 REM ACTION 3
.
.
```

This is obviously useful for menus etc.

INSTR(\$1,\$2[,sp])

(inS)

This function searches for string \$2 within string \$1, starting at position sp in \$1, or position 1 if sp is not specified. If \$2 is a null string ("") then sp (or 1) is always returned. If \$2 is longer than \$1, or the search starts beyond the end of \$1, then 0 is returned.

INSTR is useful for splitting an input into its component words in a string etc.

e.g.

```
10 INPUT "A SENTENCE";A$
20 P=0:N=-1:REPEAT:N=N+1
30 P=INSTR(A$,"THE",P)+1:UNTIL P=1
40 PRINT "CONTAINED 'THE'";N;"TIMES"
```

PRINT INSTR("BC BASIC","AS") prints 5

PRINT INSTR("BC BASIC","B",3) prints 4

INDEX AND QUICK-REFERENCE TABLES OF COMMANDS AND FUNCTIONS

CHAPTER INDEX

CHAPTER	PAGE	CHAPTER	PAGE
1	3	8	47
2	8	9	55
3	16	10	62
4	24	11	64
5	26	12	70
6	29	13	73
7	40	14	77

COMMANDS

nvf = number/variable/function. exp = expression

FORMAT	ABBR.	FUNCTION	CHAPTER
invf=z }			
!(exp)=z }		4-byte POKE z	13
'nvf=z }			
'(exp)=z }		2-byte POKE z	13
@		User def'd command	13
ADSR v,a,d,s,r	aD	Define ADSR envelope	7
AT y,x		Move cursor	1
ATTACK v	atT	Start ADSR envelope	7
BORDER col	bO	Set border colour	1
CH1,z		Disable/enable voice 3	8
CHARSET[ch][,sc]	chA	Position ch.set/screen	2,6
CLG		Clear HR screen	6
* CLR	cL	Clear vars/loops/procs	10
COPY[#cs],[n[TO n2]	coP	Copy chars to UDG RAM	2
CUTOFF cu	cU	Set cutoff frequency	8
DEFPROC name [(parameters)]	dEprO	Def. start of procedure	5
DEFBPR s,r,data	dESpr	Def. row of a sprite	3
DEFUSR char[;row;]/[,n	dEuS	Def. UDG character	2
[,n2..]			
DRAW [x1,y1] TO x2,y2	dR	Draw line on HR screen	6
[,gcol]			
ELSE statements	eL	Part of cond. structure	10
FILTER n	fI	Def. which v's to filter	8
FMODE	fM	Def. filter mode	8
GCOL n	gC	Set default plot. method	6
* GOTO(exp)	gO	Computed GOTO	14
* GOSUB (exp)	goS	Computed GOSUB	14
HIMEM=n	hI	Set top-of-RAM ptr.	2,13
HPRINT AT y,x;exp[;exp...]	hP	HR print	6
* IF cond THEN st1;ELSE st2		conditional structure	10
INK col		Set printing colour	1
KEY n,"defn"	kE	Def. function key	11

KEYLIST	kELI	List func. key defns.	11
LOCAL variables	loC	Def. local vars in proc.	5
MODE m[,s]	mO	Set display mode	6,12
MSAVE s,e[,]["fn"[,dev [,sa]]]	mS	Save block of memory	13
PAPER col	pA	Set backgnd 0 colour	1
PITCH v,n	pI	Set pitch of SID voice	7
PLOT x,y[,gcol]	pL	Plot HR point	6
PMODE m,s	pM	Set HR plot. mode/screen	6
PORT n[TO n2],z}			
PORT=n }	poR	Output to user port	11
PORT n	poR	Set DDR of " "	11
* PRINT [exprs.]	?	Output to scrn.	1
PROC name [(parameters)]	prO	Call procedure	5
PWM v,n	pW	Set pulse width of v	7
RELEASE v	reL	Release ADSR envelope	7
REPEAT	reP	Start REPEAT/UNTIL loop	10
RESONANCE n	resO	Set SID filter resonance	8
* RESTORE [exp]	reS	Reset DATA ptr.	14
RINGMOD v,z	riN	Set ring modulation of v	8
* RUN[n]	rU	Start program	10
SCRWAIT	sC	Wait for scr. blanking	6,12
SETCOL n,col	sE	Set HR/backgnd cols.	6,12
SOUNDOFF	sO	Clear SID regs.	7
SPRCLR	sprCL	Clear VIC spr regs.	3,9
SPEXP n [TO n2],zx,zy}			
SPREXP=nx,ny }	spreX	Set sprite expansion	3
SPRGCOL n,col	sprgC	Set sprite multi-col.cols	9
SPRGO n[TO n2],z}			
SPRGO=n }		Start sprite movement	9
SPRINK n[TO n2],col		Set sprite cols.	3
SPRMODE n[TO n2],z}			
SPRMODE=n }	sprmO	Set spr. multi-col. mode	9
SPRON n[TO n2],z}			
SPRON=n }		Set spr. enable reg.	3
SPRPAPER n[TO n2],z}			
SPRPAPER=n }	sprpA	Set spr/backgnd priority	9
SPRPOINT n[TO n2],ptr	sprpOI	Set sprite pointers	3
SPRPOS n,x,y		Set sprite position	3
SPRSTOP	sprst	Stop spr. movement	9
SPRTI n[TO n2],ti		Set spr. timeouts	9
SPRV n[TO n2],vx,vy		Set spr. velocities	9
SPRX n,x		Set spr. x pos	3
SPRY n,y		Set spr. y pos	3
SYNC v,z	syN	Set osc. sync.	8
UNTIL cond	uN	End of REPEAT/UNTIL loop	10
VOLUME vol	vO	Set SID O/P vol.	7
WAVEFORM v,w	waV	Set SID waveform	7

* indicates command/function present in normal 64 BASIC which has been modified.

FUNCTIONS

FORMAT	ABBR.	FUNCTION	CHAPTER
!nvf)		4-byte PEEK	13
!(exp))		Converts hex-dec	2,13
\$(+/-) hex digits		Convert binary-dec	2,13
%(+/-) binary digits			
'nvf)		2-byte lo-hi PEEK	13
'(exp))			
@nvf)			
@(exp[,nd[,z]])		Convert dec-hex	13
BIN\$(exp[,nd[,x,y]])	bi	Convert dec-binary	13
CH3		Read osc. 3	8
CHARSET(char,row)	chA	Read character set	2
CODE(n or \$)	coD	Convert ASCII-POKE code	1
ENV3	enV	Read env. 3	8
GCOL(y,x)	gC	Read scrn. cols.	1,6
HIMEM	hi	Read top-of-mem. ptr.	13
INSTR(\$1,\$2[,sp])	inS	Search for \$2 in \$1	14
JOY(j[,b])	jO	Read joystick j	4
JOYX(j)	jOx	X-movement in joystick	4
JOYY(j)	jOy	Y-movement in joystick	4
KEY	kE	Read ASCII code of key	4
KEY(c)	kE	Scan individual key	11
KEY\$	kE\$	Read key - return as \$	4
KEY\$(n)	kE\$	Read defn. of func. key	11
PADDLE(p[,b])	paD	Read games paddle	11
POINT(x,y)	poI	Test point on HR screen	6
PORT[(n)]	poR	Read user port	11
PORT [(n)]	poR	Read user port DDR	11
SCR(y,x)		Read scrn. data/HR cols	1,6
SPRDEF(s,row)	sprDE	Read line of spr.defn	3
SPREXPX[(n)]	spreXx	Read sprite X expansion	3
SPREXPY[(n)]	spreXy	Read sprite Y expansion	3
SPRGCOL(n)	sprgC	Read spr. multi-col. cols	9
SPRINK(n)		Read sprite cols.	3
SPRMODE[(n)]	sprmo	Read sprite m/col. mode	9
SPRON[(n)]		Read sprite enable reg.	3
SPRPAPER[(n)]	sprpA	Read spr-bkgnd collisions	9
SPRPOINT(n)	sprpoI	Read sprite pointers	3
SPRSR[(n)]		Read spr-spr collisions	9
SPRX(n)		Read spr. X-pos.	3
SPRY(n)		Read spr. Y-pos.	3
* VAL(\$)	vA	Conv. \$ to number	13

Comments or queries about the BC BASIC program or manual should be addressed to:

BC COMPUTERS, 31A GROSVENOR AVENUE, LONG EATON, NOTTINGHAM

Marketing and dealership enquiries to:

KUMA COMPUTERS LTD., 12 Horseshoe Park, Pangbourne, Berks
Telephone 07357-4335 Telex 849462 Prestel *2473222#

